NISTIR 5917

Device Driver Development for Microsoft Windows NT[1]: Accessing Motion Control Hardware Using a Multimedia Framework

Richard D. Schneeman
Computer Scientist
rschneeman@nist.gov

U.S. Department of Commerce
Technology Administration
National Institute of Standards and Technology
Automated Production Technology Division
Manufacturing Engineering Laboratory
Gaithersburg, Maryland 20899 USA

NIST

ABSTRACT

In order to integrate motion control hardware into an advanced operating system environment such as Microsoft Windows NT, a flexible set of device drivers delivering a wide range of motion control services are required. This paper describes the set of user and kernel-mode Windows NT device drivers NIST researchers developed to provide device-independent software-based access to motion control hardware in support of various research activities underway here. A technical presentation of the device driver design and development process are given to benefit those designing or in the process of migrating device drivers to the Windows NT environment. As an important by-product of this device driver effort, a motion control extension to the *defacto* standard Microsoft Media Control Interface was developed to provide device independent access to motion control hardware. This capability places motion control hardware in the same category as other Microsoft Multimedia compliant computing peripherals such as sound, audio, video, and compact disk devices.

Keywords: application programming interfaces, device drivers, device independence, interrupts, kernel-mode, motion control, multimedia, operating systems, portability, user-mode.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

Motion control refers to the automated methods by which an application can control hardware axes, coordinate systems, motors, and input/output data in a specific manufacturing domain. For example, the majority of motion control applications are found in rather mundane and discrete ***transfer-line*** style applications, such as controlling the operations that paint M&M's or that affix labels to soup cans. However, there are other highly specialized and complex industrial applications as well, ranging from controlling large machines to orchestrating robotic movement. As part of ongoing manufacturing research efforts at the National Institute of Standards and Technology (NIST), the need to host generic motion control hardware using an advanced operating system environment such as Microsoft Windows NT was required.

In order to provide motion control-based application access to the hardware, device driver software is needed to drive the specific motion control board from within application-space. Unfortunately, device drivers for the Windows NT environment did not exist. In association with Delta Tau Data Systems (Delta Tau) a Cooperative Research and Development Agreement (CRADA) partner from industry, researchers from the Manufacturing Engineering Laboratory (MEL) at NIST have developed an initial Windows NT-based motion control device driver platform to serve as the basis for several manufacturing research projects. Delta Tau develops and markets a product specifically for the motion control domain called a Programmable Multi Axes Controller, or PMAC. PMAC essentially functions as a plug-in central controller of motion peripherals for an Industry Standard Architecture (ISA) BUS-based computer. The ISA board for the Intel 80x86 class of microprocessors was used for this study.

A key aspect of this device driver development effort was to provide a means of accessing ***any*** motion control hardware, not just PMAC, from within an application using hardware independent methods to ensure application portability. This was accomplished by using a standardized messaging scheme based on the Microsoft ***defacto*** standard Media Control Interface (MCI) specification[2]. This common messaging scheme developed as an extension to MCI provides access to the Motion Control domain through a standard Multimedia-based application interface. This portable interface to motion control provides exciting possibilities for mainstream multimedia-based applications.

The use of a MCI-based multimedia device driver will provide developers with hardware independent means to generate motion control-based applications regardless of the underlying motion control board used. Motion control vendors could support these standard interfaces in the form of libraries that would provide standard application access to their motion control boards. This article addresses the application programming interfaces and the device driver software development efforts undertaken using the Microsoft Windows NT operating system. It is important to note that the device driver design methodology used during this study would be appropriate for other motion control board vendors as well. A majority of the device-dependent kernel-mode layer will also be usable to vendors as many support similar techniques for board communication schemes. We simply used the motion control board developed by Delta Tau because it was readily available on a machine that had previously been used in another study.

[2] MCI is the Microsoft sanctioned *defacto* standard method for accessing multimedia devices. By extending MCI to include motion control, we provide yet another standard multimedia accessible peripheral interface.

1.1 NIST NT Device Driver Design

Within the Windows NT operating system, software executes in one of two modes — user-mode or kernel-mode. User-mode is essentially a non-privileged processor mode in which all access to system data is allowed only through well-defined subsystem-supplied functions, which in turn call system services. User applications, the Win32 subsystem, and some device drivers execute above the line in the user-mode realm. User-mode device drivers do ***not*** have access to hardware; therefore, they must call a kernel-mode supplied device driver. Kernel-mode processes utilize a privileged processor mode in which the Windows NT Executive (kernel) and kernel-mode device drivers execute. A driver or thread running in kernel-mode may access system memory and physical hardware directly.

The device driver architecture we designed is based on the idea of providing a paired set of user-mode and kernel-mode Windows NT device drivers to bridge the two system modes. This paired set would provide both a device-independent user-mode driver to allow application portability, while also maintaining a device-dependent kernel-mode driver for low-level hardware access, and to provide a centralized area for hardware specificity. Figure 1 illustrates the NIST Windows NT user-mode and kernel-mode device driver architecture.

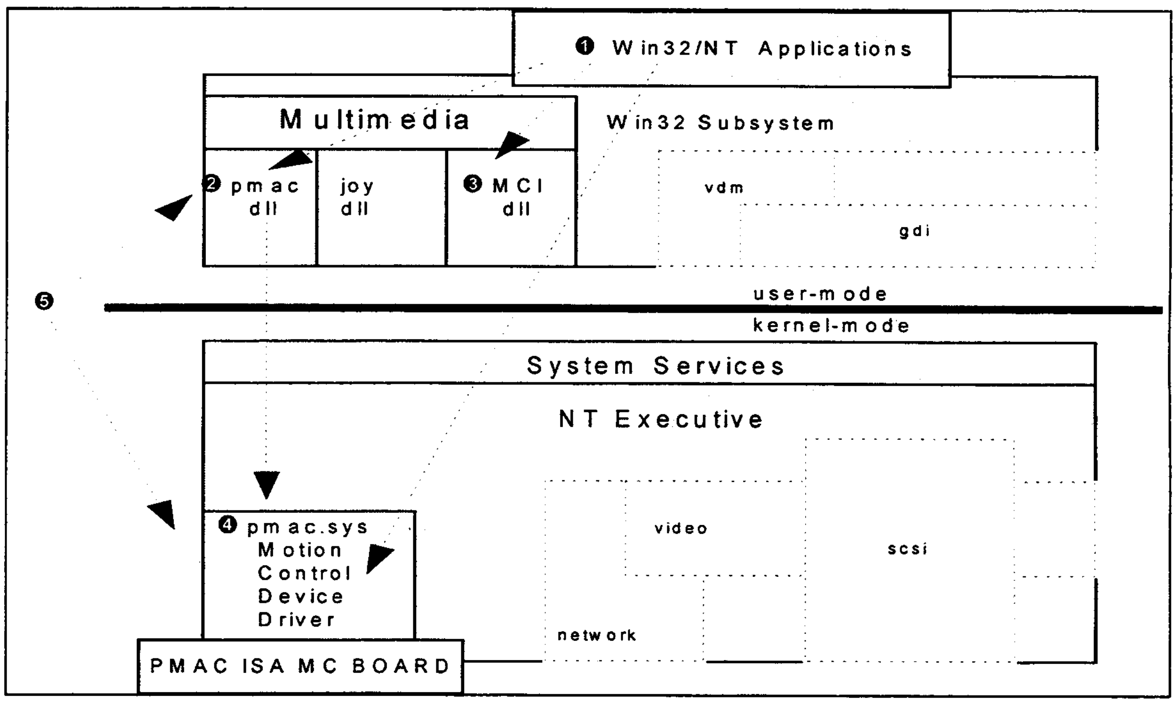

Figure 1: The NIST Windows NT Device Driver Structure.

The thick dark line shown in Figure 1 separates applications, subsystems, and devices drivers into either user-mode or kernel-mode processor status areas. The NIST developed user-mode device driver called ***pmac.dll***, shown in area ❷ consist of a Win32-based user-mode multimedia device driver that provides Windows NT application-level access in area ❶ with motion control related services.

Using NIST developed custom extensions to various multimedia application programming interfaces (API) provided by Windows NT, we implemented ***pmac.dll*** using a standardized interface for motion control that was based on the MCI specification[3]. A Win32 application would then obtain multimedia services by using either the high-level device-independent MCI or use a low-level device-independent API, both of which are embedded into the ***pmac.dll*** device driver. The user-mode device driver used in the device driver architecture design satisfies the NIST interface concerns for allowing application portability and interoperability.

Area □ illustrates the path an application uses to call MCI directly for basic or required system commands that fall outside the NIST developed MCI command set in area ❷. The second device driver in the paired set includes ***pmac.sys***, this is the Windows NT kernel-mode device driver, shown below the line in area ❹. Notice the dotted arrows showing the relative direction of the messaging and application calling paths. When the user-mode device driver in area ❷ requests hardware access in response to an applications request from area ❶, the user-mode driver must call down to the kernel-mode driver in area ❹ to perform all hardware-level access on behalf of the user-mode driver.

The calling path leading directly from the application to the kernel-mode driver represents an emulated interface to the legacy application/DLL (dynamic-link libraries) environment. We provided this parallel interface implementation in support of legacy applications and DLL's that we wanted to port into the NT/Win32 area later. That is, when a DLL associated with the legacy applications requires hardware access, it calls the kernel-mode driver directly using a Win32 API function call ***DeviceIoControl()***, or by directly addressing portions of memory-mapped RAM, that were previously mapped into the user's address space. This method bypassed the user-mode device driver in order to maintain the backward compatibility with the legacy driver interface. Area ❺ shows the complete NIST Windows NT paired device driver set.

Before specific implementation details of the user-mode and kernel-mode device drivers can be discussed, it first becomes necessary to describe the Windows multimedia architecture. The Windows multimedia architecture provides the application interaction model in which to support the device independent user-mode device driver. The device dependent hardware specific areas were supported using a kernel-mode Windows NT device driver. The two device drivers working together would satisfy all application requests for motion control related services. Therefore, all application interaction with the device driver framework as we have designed it, must take place at the user-mode multimedia-based Windows NT device driver level. The next section provides a background on the Microsoft Windows NT multimedia architecture, including an overview of the Media Control Interface (MCI) used to implement the custom message-based user-mode multimedia device driver.

---

3 API refers to application programming interfaces. In this context, API can either mean an actual function call signature, or the term can refer to the actual standardized messages sent as part of the MCI interface.

2. Windows NT Multimedia Architecture

As background for the multimedia driver implementation, this section briefly discusses the Windows NT multimedia architecture. The Windows NT Multimedia architecture design philosophy centers around two very important concepts: extensibility, and device independence. It is because of this design philosophy that the multimedia architecture can readily accommodate new technology without modification to the architecture. The multi-level, device-independent methods for accessing multimedia resources provide applications with different levels of multimedia support while also providing the capability of execution on a wide variety of hardware platforms. Essentially, all extensibility and device-independence in the Windows Multimedia architecture is provided by three layers of software subsystems including: the WINMM multimedia library, actual device drivers for multimedia, and MCI device drivers. Each of these subsystems will briefly be described.

2.1 The Windows Multimedia Library

Windows provides an interfacing layer between applications and device drivers — applications do not call device drivers directly. When an application calls a function that requires a device context, Windows uses the WINMM library to translate the call into a message and then dispatches that message to the appropriate device driver. The WINMM library provides this translation layer to insulate applications from the device drivers while centralizing the device independent code. In addition, the run-time linking that provides this translation layer with the necessary device drivers it needs is included in the WINMM module. WINMM library also provides dispatch capabilities for high-level MCI services and the low-level multimedia support functions. When the inverse relationship occurs, that is a device driver must notify a client of an event, it calls the WINMM library with a message that is routed back to the client application. The term client application is sometimes used to describe the Windows interface from the device driver's perspective. The client application can include the Windows environment as well as the application that is controlling the multimedia operations.

2.2 Multimedia Device Drivers

Multimedia device drivers and other installable drivers are dynamic-link libraries, usually written in C or assembly language, or a combination of the two languages. Multimedia device drivers provide the communication between the low-level WINMM library functions and multimedia devices. These drivers present a well-defined and consistent driver interface to minimize device dependent cases while providing the user with error free upgrades and installation of future device drivers in the system.

All Win32-based multimedia device drivers have essentially the same functional structure — they provide exported functions to process messages received by the driver. There is one common function that centralizes all the message reception activities, called ***DriverProc()***[4]. This function is

4 When a dynamic link library is compiled and linked into the system, the function signatures or API, i.e., *DriverProc()*, must be visible to the application programs if it is to be used. The export process allows the developer to selectively choose which functions should be globally visible to the user. By doing selective exports, the developer can shield the DLL from unwanted abuse of internal functions that are not a part of the usable interface.

called by the Windows system when a request is made to install, configure, or to use the device driver. ***DriverProc()*** handles all requests to install, open, close, remove, and configure a given device. It *must* be provided if the driver supports configuration by the user. There is a set of standard system messages that the driver must be able to handle in response to requests from an application or from the Control Panel Drivers applet of the Windows NT graphical user interface[5]. Some of these include: DRV_OPEN, DRV_CONFIGURE, and DRV_LOAD. The ***DriverProc()*** entry point processes messages sent by the system to the driver as the result of an application call to a low-level function. For example, when an application opens a motion-control device, the system sends the specified device driver a DRV_OPEN message. The ***DriverProc()*** function receives and processes this message in the device driver. Unsupported messages are handled using a default processing module inherent to the Microsoft multimedia architecture.

The DRV_CONFIGURE message is sent to the device driver when the user or another application wants to configure the device for installation and use. The NIST multimedia device driver supports more than one interrupt-level, I/O port assignment, and Dual Ported RAM address; therefore, it provides a configuration menu in the Drivers section of the Windows NT system Control Panel applet. Using the Control Panel to configure the driver automatically forces the WINMM subsystem to send a DRV_CONFIGURE message to the installable driver. The driver, after it receives the configure message, must display a configuration dialog box for this purpose. The dialog box is inherently a part of the device driver code and displays the name, version number, and other information about the device driver in addition to the user-configurable fields for modifying the PMAC motion control board. The initialization information obtained by the driver during the configuration phase is subsequently stored in the Windows NT Registry for later probing by the kernel-mode driver initialization routines[6].

2.3 MCI Device Driver Overview

Device drivers for MCI provide the high-level services for the control of multimedia peripherals and devices. The MCI provides applications written for the Microsoft Windows environment with device-independent capabilities for controlling various multimedia devices such as animation players, videodisk players, and audio hardware. MCI provides an extensible interface to device-independent control of virtually any peripheral device. By using the Microsoft sanctioned methods for *extending* MCI, we were able to develop an MCI interface for motion control devices, such as the PMAC motion control board from Delta Tau.

MCI device drivers are in fact Windows multimedia device drivers. MCI-based Win32 multimedia drivers accomplish this by extending the ***DriverProc()*** entry point to accept MCI messages specific to that device type or by adding additional entry points by selectively exporting their custom multimedia API. Because motion control is new to the MCI, we needed to create a completely new set of MCI commands for motion control. In addition to the MCI interface, we supplemented the multimedia device driver with additional exported entry points for a low-level motion control API. We added both capabilities because we wanted to provide a device-independent and versatile interface to control motion peripherals, while also providing an additional robust, data-

[5] Applet refers to a Windows-based graphical user interface menu, dialog box or similar user input display area.
[6] Registry (or registry hive) is a Windows NT and Windows 95 maintained database that holds all software and hardware configuration information in the system. The information is used frequently in the *Plug-and-Play* subsystem.

driven capability to the architecture. We provided the *standardized* extensions to the ***DriverProc()*** entry point using motion-related MCI messages and exported functions[7].

Microsoft encourages multimedia software developers to use and to develop with the MCI interface instead of hardware specific functions because of MCI's device independent characteristics. By using Microsoft's methods for extending the MCI, we have developed a first draft specification and implementation of motion control for industry review. If adopted as a primary MCI-based method for controlling motion devices, it could benefit the industry by providing low-cost, universal coverage of an application programming interface for generic motion control. An overview of the types of MCI devices and the programming interfaces available is given in the next section in order to familiarize the reader. This background will be helpful when the actual MCI/multimedia driver implementation is discussed.

2.3.1 MCI Devices

In the MCI domain, devices can only take one of two possible forms — simple or compound. *Simple* devices range from compact discs (CD) to videodisk players, while *compound* devices require a data file for playback conditions. In the latter case, a data file associated with the device is called a *device element*. MIDI files for music, WAVE files for audio, and AVI files for video are all examples of compound device elements. The NIST MCI driver is only concerned with simple devices — motion control devices.

Device types identify standard classes of MCI devices, such as videodisk players, CD audio devices, or image scanners. *Device names* are used to uniquely represent a specific device driver instance (i.e., pmac0), since there may be several drivers for one particular device type available at any given time on the system. The device names are located in the MCI32 part of the Registry on a Windows NT machine. The device names are sequentially numbered based on whether it is the first, second, etc., driver installed on that system for a particular device class. To use a MCI device from within an application, the *simple* motion control device is opened by specifying the device name in the ***mciSendCommand()*** call shown in Figure 2. The *mciOpenParms* data structure in Figure 2 provides a place holder for the specific device type. In our case, we use the "pmac0" *device name* as the identifier to indicate a motion device. Either a device name or device type can be used to specify a device during the open. In our device driver implementation however, the call to open "pmac0" will translate into and specify the device type MCI_DEVTYPE_OTHER. This is a special device type and is used because the motion device type is undefined in the Microsoft multimedia architecture at this time. All devices falling outside of the currently defined device list in the Microsoft multimedia architecture must use this designation to disambiguate between open calls. Besides opening the device, there are also corresponding commands to close the device, query the device status, and to find out about specific device capabilities.

```
MCI_OPEN_PARMS      mciOpenParms;
DWORD               dwStatus;
        mciOpenParms.lpstrDeviceType = "pmac0";
        dwStatus = mciSendCommand( NULL, MCI_OPEN, ... );
```

Figure 2: Specifying the Device Name in the MCI Open Device Call.

[7] The term *standardized* used here refers to the Microsoft sanctioned formats, structures, and methods that we used to define and extend the MCI-based command set and API for the control of Motion Devices.

2.3.2 MCI Programming Interface

MCI provides two application programming interfaces for communicating MCI specific information to the driver: an ASCII string command interface called a *command-string* interface and a message-based interface termed a *command-message* interface. The command-string interface allows users to embed English-language string commands into their applications in order to communicate with MCI devices. Using a text-based interface for example, assume a multimedia application requires the use of a videodisk player to display video frames on the screen. The application could use MCI to issue a request to place the currently selected videodisk player in play mode, and to display the consecutive video frames from 1 to 10 by embedding the following text into the application: ***"play videodisk from 1 to 10".*** Text-based interface applications, high-level languages such as Microsoft Visual Basic, and multimedia authoring environments all use this type of interface to provide device control to their respective application domains.

The command-message interface on the other hand provides a message-passing paradigm to communicate with MCI devices. Applications that require a C language binding to directly control MCI devices can use the command-message interface. In addition, the message-based paradigm allows a smoother integration of multimedia software into event-driven Windows-based applications. If desired, it is also possible for C language-based applications to use the command-string interface. To do this from within their application, developers use the ***mciSendString()*** API function together with the specific MCI English-language statements embedded as a quoted parameter to the function. By design, MCI device drivers are able to handle both the string and message-based invocations. The code segment in Figure 3 illustrates the data structure and MCI API ***mciSendCommand()*** function call that performs the identical operation as the string invocation using the command-message interface.

```
WORD                wDeviceID;
MCI_PLAY_PARMS      mciPlayParms;
DWORD               dwStatus;

        mciPlayParms.dwFrom = 1L;
        mciPlayParms.dwTo   = 10L;
        wDeviceID = mciGetDeviceID( VIDEODISC );
        dwStatus = mciSendCommand( wDeviceID, MCI_PLAY, (MCI_FROM | MCI_TO), &mciPlayParms );
```

Figure 3: A C Language Binding for the *mciSendCommand()* Videodisk Example.

In general, the command-message includes the following items: **(1)** a constant command message value, i.e., MCI_PLAY, **(2)** a data structure that acts as a container for additional parameters used in the call, i.e., *mciPlayParms*, and **(3)** a set of flags that specify what items in the data structure are enabled or *validated* (i.e., MCI_FROM). The commands are sent using the ***mciSendCommand()*** API function call, which includes the parameters for the device ID, message, flags, and a pointer to the data structure. The function call returns an error string associated with an error number (returned in the *dwStatus* variable). WDeviceID is the Id associated with the current device retrieved using the ***mciGetDeviceID()*** function. MCI_PLAY is the command-message command, MCI_FROM and MCI_TO constitute the logically OR'd flags to notify MCI that they are being used and thus should be validated upon receipt, and &mciPlayParms is an address to the data structure holding the FROM and TO frame intervals as well as other possible parameters. The next section discusses the NIST multimedia device driver from a design and implementation approach. A specific example of integrating the device driver into the WindowsNT multimedia architecture is also described in detail.

3. User-Mode Multimedia Device Driver Design

In the previous sections, a general overview about the Windows Multimedia Architecture including the MCI interface was given as background. In this section a more detailed look at the interworkings of MCI device drivers are highlighted with specific reference given to the NIST multimedia-based motion control device driver. This section illustrates specific ideas that are relevant to *all* MCI device drivers including the message format, data structure layout, flag determination and usage, and the command table composition process. However, they are presented here in the context of motion control using examples from the NIST MCI device driver.

Using the Windows multimedia architecture, both high and low-level services can be bridged together using a consistent interface to provide a wide range of service capabilities and functionality. The use of special device drivers that interpret and execute MCI commands are essential for providing the device independence. These drivers can control the hardware directly or through the low-level functions provided by the Windows Multimedia library WINMM. Our Win32 MCI driver provides the *hooks* for both the indirect hardware control using MCI and a homegrown low-level Windows API for direct control of hardware — although not currently an integral part of the Windows multimedia definition[8]. Therefore, we developed both the low-level motion control functions as an API provision, and an MCI-based command set while also integrating them into the currently defined Windows architecture for multimedia. Figure 4 illustrates where the NIST high-level MCI interface and low-level API functions for motion control reside in the currently defined Windows multimedia architecture[9].

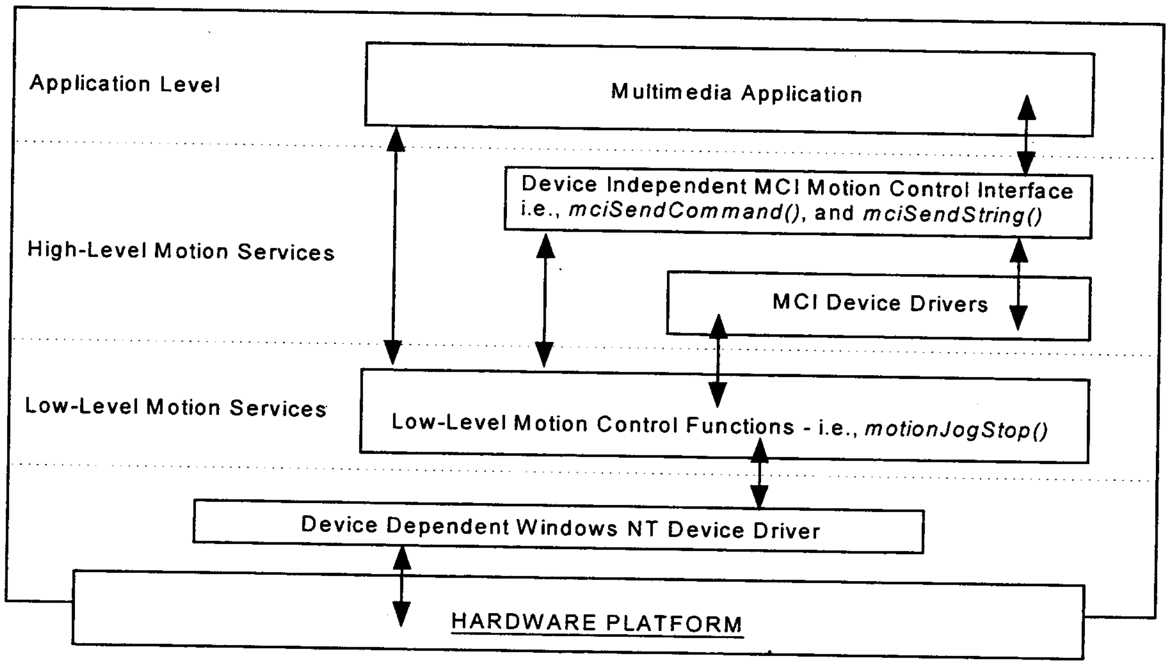

Figure 4: Windows Multimedia Architecture Augmented with Motion Control.

8 Microsoft has not yet developed a Windows-based Multimedia/MCI solution for Motion Control.

9 The multimedia architecture depicted here is based on the Windows Multimedia architecture as described in the Microsoft Win32 Programmer's Reference, Volume 2. The architecture has been modified to include placement of the NIST Motion Control device drivers.

Figure 4 illustrates the various paths a multimedia application can follow to obtain motion control related services. The first method (multimedia application to MCI) uses the high-level MCI interface directly to control motion devices. The second method (multimedia application to motion control functions) uses the low-level API functions to manipulate the device using such API functions as ***motionJogStop()***. Direct use of the low-level API from an application level instance may be required to exploit extensive device capabilities that may not be provided by the higher-level MCI services interface such as those requiring a data transfer capability. Notice the path from the MCI interface (high-level) to the low-level motion control functions; in this case, MCI may need to use the low-level services to support its interface. The low-level API is provided for such applications as a motion control tool package — or software that would otherwise require specialized motion requirements.

3.1 High-Level MCI Application Programming Interface

MCI device drivers are Windows Multimedia drivers; therefore, they must follow the standard Windows multimedia approach of providing a ***DriverProc()*** entry point. In this section, we describe how MCI messages are defined with examples of how this entry point handles the new MCI message context. MCI uses this entry point to handle messages for MCI services and system messages that install, initialize, and configure the driver. The changes made to the entry point during the development process are based on custom messages designed for the particular device being used. The developer augments the ***DriverProc()*** entry point to handle these new messages. The construction of motion control messages uses the Microsoft standardized conventions.

MCI can use up to four types of messages including required, basic, extended, and system messages. We did *not* extend any existing MCI command sets; this was impossible due to MCI currently not supporting motion control. However, we did extend the MCI framework in general to include motion control by creating a completely new command set that augments the other four message types. In addition, some subsets of the basic and required commands were provided such as OPEN and CLOSE. In the following section we discuss how these new messages are developed and integrated into the existing high-level MCI multimedia infrastructure.

3.1.1 Adding New Commands to the MCI System

Motion control is not currently a device type that is controllable using the standard Microsoft MCI component interface API. For this reason, we were not able to simply extend an existing command table for our use. Instead, a completely new set of commands for the motion control domain were developed. Microsoft provides a methodical way to create and integrate new MCI commands into the multimedia architecture. The following steps have been summarized below:

1. Define the new MCI custom command-messages that the driver will support.
2. Define the new data structures and flags in support of these new commands.
3. Generate a custom command table for the MCI parser to use for command translation.
4. Provide support for these new messages in the driver's ***DriverProc()*** entry point.
5. Register the custom command table with MCI during the driver's DRV_LOAD phase.

These steps will be covered below with respect to the NIST multimedia device driver implementation.

3.1.1.1 Defining the Custom Messages

In a separate C language header file, all new messages to be processed by ***DriverProc()*** should be defined with a specific command name and an unique message identifier. For all custom or new messages defined in this header file, their message identifiers should be offset by the constant MCI_USER_MESSAGES. This special constant is the first integer available for user-defined custom messages, and insures that the new starting identifier sequence does not overlap any known messages within the standard MCI command space. The following code segment in Figure 5 illustrates the message definition area of the header file that we used in our MCI device driver. Notice the highlighted command-message MCI_SERVO_JOG; we will expand on this message and use it in an upcoming example to illustrate how new MCI messages are built and used.

```
/************************************************************
 * MCI Device-Specific Extended MCI Message Commands
 ************************************************************/
#define MCI_COMMAND_FIRST       (MCI_USER_MESSAGES + 1)
#define MCI_SERVO_JOG           (MCI_COMMAND_FIRST)
#define MCI_SERVO_HOME          (MCI_COMMAND_FIRST+1)
#define MCI_SERVO_DISABLE       (MCI_COMMAND_FIRST+2)
#define MCI_INTR_INIT           (MCI_COMMAND_FIRST+3)
#define MCI_INTR_TERM           (MCI_COMMAND_FIRST+4)
#define MCI_COMMAND_LAST(MCI_COMMAND_FIRST+50)
```

Figure 5: Extending the MCI Command Set For Motion Messages.

A core MCI command set was identified and used as a starter set for motion control based messages in this driver. Core commands that JOG (rotate the axis of a motor to a specific point) and HOME (bring the motor axis back to its original starting position) the axis of a motor are representative of the typical operations a motion control board would provide. Notice the messages are offset by MCI_USER_MESSAGES to alleviate confusion during processing by MCI.

3.1.1.2 Defining the Data Structures and Flags

After the custom messages such as MCI_SERVO_JOG have been defined in our header file, an associated data structure for that message also needs to be defined. Any MCI message sent to ***DriverProc()*** contains two parameters, *lParam1* which consists of a concatenated flag field that describes which elements of the data structure have been enabled, and *lParam2* which contains the pointer to the data structure. For each member of the data structure that provides data from the calling application, there must exist an associated flag. The application uses the flags to bit-wise OR together the flags field representing the *lParam1*. The flags may also specify options without necessarily corresponding to any associated member of the data structure. Each member in the data structure is based on a Windows 32 bit DWORD or a multiple of a DWORD. There is no limit on the number of members in the structure; the amount of members declared is based on the type of message being described. There are certain restrictions on what must be the first member of the structure and what types of data must be associated with certain return members. For instance, referring to our MCI header file example once again, notice the data structure we defined below in Figure 6 for the MCI_SERVO_JOG command message.

```
typedef struct {
        DWORD           dwCallback;
        DWORD           axisInt;
        DWORD           speedStr;
        DWORD           szSpeedStr;
} MCI_SERVO_JOG_PARMS;
typedef  MCI_SERVO_JOG_PARMS *LPMCI_SERVO_JOG_PARMS;
```

Figure 6: MCI_SERVO_JOG Data Structure.

A structure of type MCI_SERVO_JOG_PARMS will be allocated prior to each command-message function's use of the structure. The first member of the structure *must* be reserved for a handle to a callback function that MCI uses for the MCI_NOTIFY flag. MCI provides the MCI_NOTIFY flag for developers to use if application-level notification of the command status is required. After an application issues a command to the MCI system, asynchronous behavior within the application can be maintained by forcing MCI to immediately relinquish control in the driver to the application. In this case, the application may want to be notified later after the execution of a particularly lengthy command has in fact completed. If on the other hand a more synchronous behavior is required whereby the application must *hold up* until the device actually completes the command, then the MCI_WAIT flag is provided. During the wait operation, user input is not monitored; therefore, MCI supports a break key definition to re-establish communication with the applications main-loop in the event the notification does not propagate (due to problems with the physical device).

Every data structure defined for use with MCI *must* contain a field for a *dwCallback* element. This element provides a storage area for the application's window handle assignment for use in the notification process. Notification to the application will be posted to the application's main window messaging loop using the MM_MCINOTIFY message after the device command completes. The low-order word (16-bits) of the MM_MCINOTIFY message will contain the specifics of the notification (i.e., successful, aborted, failure, or superseded). When the application needs to be notified (i.e., a MCI_NOTIFY flag was enabled), the driver must issue a ***mciDriverNotify()*** at some specific time — typically when the command has been completed[10]. The driver, in response to a MCI_NOTIFY flag from the application should do the following tasks:

1. Determine if the MCI_NOTIFY flag has been set.
2. Configure a timer or event to fire off when the specific device has finished the task.
3. Return control to the caller pending notification.
4. When the task completes, notify the application-specified callback via ***mciDriverNotify()***.

Returning to the data structure example in Figure 6, other requirements for data members are placed on the structure if the structure is to return data. If no data is returned in the call and only data members are needed to pass information on to the driver then no data members need be reserved. This is the case in the example shown in Figure 6. The first data member is the *dwCallback* DWORD; this member must be present and is mandatory for MCI. The next field is the *axisInt* variable indicating an integer is stored there that will represent a motor axis if the

[10] In general, the conditions and timing of the notification back to the application is completely device driver dependent. In fact, notification may return immediately, after the actual command completion on the physical device, or at some other pre-determined driver interval.

corresponding flag has been set. A partial list of MCI message flags are shown in Figure 7. These flag definitions have also been defined within the MCI header file used for the command data structures. The flags are used to indicate which data members (such as the *axisInt*) are currently being used. In addition to specifying data members, the flags are also used to indicate an option such as the MCI_NOTIFY or the synchronous MCI_WAIT flag option.

```
/************************************************
 * MCI Device-Specific Extended MCI Message Flags
 ************************************************/

#define  MCI_SERVO_AXIS            0x00010000L
#define  MCI_SERVO_SPEED           0x00020000L
#define  MCI_SERVO_STOP            0x00040000L
#define  MCI_SERVO_POS             0x00080000L
#define  MCI_SERVO_NEG             0x00100000L
#define  MCI_INTR_MASK             0x00200000L
#define  MCI_INTR_HAND             0x00400000L
.......... ...............................  ...................
```

Figure 7: MCI-Specific Message Flags Used in the Driver.

Just as new MCI commands for motion control cannot conflict with any predefined MCI commands, we also need to uniquely differentiate bit fields for the new command set flags. Bits 0 through 15 of a 32-bit DWORD are reserved for use by MCI. Therefore, bit 16 (0x00010000) is the first bit pattern that can be used with the new driver flags. Unused bits in a flag must be set to zero. Again using the MCI_SERVO_JOG command as an example, when the MCI_SERVO_AXIS flag is set in the data structure, this tells MCI that a value is assigned in the *axisInt* field of the data structure.

3.1.1.3 Creating the Custom MCI Command Tables

For the most part we have discussed how the new motion control command set definition process supports the command-message interface. However, when the command-string interface is in use, there must be some way for MCI to parse the string and generate the equivalent command-message format that is ultimately used as parameters to the ***DriverProc()*** function of the MCI driver. The construction of a custom command table provides the input to the parser in table-driven format for the command-string recognition process. Typically, the table data is stored as Windows resources (RCDATA) within the driver and is bound to it during compile and link time. Command tables may also be dynamically bound to an executing driver by declaring it as external. External tables are compiled as separate Windows DLL's and can be dynamically loaded and unloaded as needed. In our implementation of the device driver, resources for the command tables were bound at link time within the executable image of the device driverDLL. In some cases, the developers may want a less constraining way of binding command tables than at link-time. The attractive feature of providing external tables is that other developers can add and extend the existing commands simply by adding their specific DLL with the new command tables. Otherwise, the developer needs access to the source code of the original device driver. In addition, developers may want to provide multi-language subsets of the commands so that internationalization of the driver is possible. For these cases, external resource files can be bound during run time so that the command tables can be found during the actual execution of the driver. This provides a flexible alternative to link-time binding. Figure 8 below illustrates a portion of our device driver command table example.

```
/**************************************************************
 * MCI command table for MCI Motion Control Device-specific commands
 **************************************************************/
pmac RCDATA
BEGIN
  L"jog\0",      MCI_SERVO_JOG, 0,       MCI_COMMAND_HEAD,
  L"notify\0",   MCI_NOTIFY,             MCI_FLAG,
  L"axis\0",     MCI_SERVO_AXIS,         MCI_INTEGER,
  L"speed\0",    MCI_SERVO_SPEED,        MCI_STRING,
  L"pos\0",      MCI_SERVO_POS,          MCI_FLAG,
  L"neg\0",      MCI_SERVO_NEG,          MCI_FLAG,
  L"stop\0",     MCI_SERVO_STOP,         MCI_FLAG,
  L"\0",        0L,                      MCI_END_COMMAND,
  ........
  L"home\0",     MCI_SERVO_HOME, 0,      MCI_COMMAND_HEAD,
  ........
END
```

Figure 8: Partial MCI Command Table for Motion Control.

Essentially, the command table consist of three columns of information. The first column includes the commands and flags that would be used in the command-string interface. Each quoted string is zero terminated and must be prefixed with an L to indicate that wide-string literals are in use. The second column contains 32-bit DWORD commands and flags that would be used to construct a command-message during the parsing phase. The third column contains 16-bit WORDS identifiers that allow the MCI parser to interpret each type of entry in the command table.

The table is aligned on 32-bit DWORDS; therefore, it was necessary to pad the WORDS in the second column to fill out the table in DWORDS. This can be seen with the command MCI_SERVO_JOG,0 example from the table. The 16-bit WORD was aligned to 32-bits by adding a zero after the command definition. Each command table entry is defined as a command table list and defines the parsing action of a particular command. For instance, the partial command table above provides several command lists which directs the parser to which flags are currently valid and how the corresponding data structure is defined for the ***jog*** command. Likewise, the command table entry for the ***home*** command would provide yet another set of lists to this command table.

Microsoft provides a rigid structure for defining the tables: each verb block of a command must begin with an MCI_COMMAND_HEAD and end with an MCI_END_COMMAND delimiter. The entire command table text must end with an MCI_END_COMMAND_LIST denoting the end of all possible commands. The tables can define return values that must be accounted for in each data structure. Return types can include strings, integers, window handles, rectangles etc. If the MCI_FLAG command flag is present in the third column, then MCI does not associate this with any member in the data structure — such as MCI_NOTIFY. Like commands, flags in the tables use the first column to represent the string command used for the flag. When the MCI parser comes upon a command list entry of either MCI_INTEGER, MCI_STRING, or MCI_RECT, then the entry will immediately be associated with a data member in the data structure. The data members are not explicitly named in the command list; however, they are positionally referenced by the parser. That is, the order in which the entries appear in the command list implies the order of the data members as they appear in the data structure. MCI reserves the first member in the data structure for the notify callback function. The second member is associated with the first entry needing space in the data structure. This filling out of data structures to entry members continues until the list is terminated. This association assumes that the data members are aligned on either DWORDS or a multiple of DWORD. Command flags which do not require referencing of data members can be interspersed throughout other table list entries. The flags will not change the alignment of the data members, they only set bits in *lParam1* of ***DriverProc()***.

3.1.1.4 Preparing *DriverProc()* for the New Messages

Now that we have described the message format, the data structures, flags, and the command tables, let's illustrate how the ***DriverProc()*** function is updated to reflect the newMCI message set. The most straightforward way to accomplish this is simply to modify the default handler in ***DriverProc()*** to look for the new range of messages we have defined. By prefixing the MCI_USER_MESSAGES constant to each new MCI message, we have effectively defined a range of new command values that can be selectively chosen.

Figure 9 illustrates how the first command value and the last command value are used to selectively determine if the multimedia driver should accept the command message as one pertaining to our MCI motion command range. In the partial ***DriverProc()*** handler shown below in Figure 9 we simply expanded the range of messages to include our newly defined MCI messages. Notice that we continue to honor the existing MCI core and required set of MCI commands by maintaining the test for the DRV_MCI_FIRST and DRV_MCI_LAST ranges. After ***DriverProc()*** has selected the new message in its default processing, it merely calls the ***motionProcessMessage()*** function in the driver to handle the specific MCI command (i.e., MCI_SERVO_JOG) that was initially issued.

```
default:
        /* select messages in the MCI range */
        if (!HIWORD(dwDriverID) &&
         uiMessage >= DRV_MCI_FIRST && uiMessage <= DRV_MCI_LAST ||
         uiMessage >= MCI_COMMAND_FIRST &&
         uiMessage <= MCI_COMMAND_LAST)
        {
                return motionProcessMessage(dwDriverID, uiMessage, lParam1, lParam2);
        }
```

Figure 9: Portion of the *DriverProc()* Entry Point to Handle Custom Messages.

The actual processing associated with each custom message is provided in other source modules of the MCI device driver. Notice in the call to ***motionProcessMessage()***, the parameters in the function include the *dwDriverId* which is the device id that was parsed, the *uiMessage* which is the MCI_SERVO_JOG message, and the two *lParam1* and *lParam2* which correspond to the flags bit field and the pointer to the data structure, respectively.

DriverProc() can now handle the new messages generated by the custom command tables. However, in order for the device driver to use the new commands, the command tables must first be registered with MCI. This process is covered in the next subsection.

3.1.1.5 Registering Command Tables with MCI

After the command tables have been setup properly and ***DriverProc()*** has been modified to handle the new messages, the command tables need to be registered with MCI so that they can be referenced when called upon. These modifications again occur in the ***DriverProc()*** function code. Typically, during the DRV_LOAD phase of the DLL loading into memory, the tables are registered with MCI. The partial ***DriverProc()*** code snapshot in Figure 10 illustrates this method by calling ***mciLoadCommandResource()***. When the device driver is opened in the ***DriverProc()*** section, we set the device type to MCI_DEVTYPE_OTHER because we are using a device that is undefined to MCI. When MCI sees the other device type designation it automatically searches for the type-specific (motion) command table that we registered. When an application has closed all of its remaining devices, the device driver is taken out of memory with the DRV_FREE call from the Windows multimedia subsystem. In the ***DriverProc()*** function where the DRV_FREE message is collected, the custom command tables should be freed as well. This is accomplished by calling the ***mciFreeCommandResource()*** function in the DRV_FREE message handler shown in Figure 10.

```
switch (uiMessage) {
    case DRV_LOAD:
        /* register the custom command table */
        LoadString(ghModule,IDS_COMMANDS,aszResource,sizeof(aszResource));
        wTableEntry = mciLoadCommandResource(ghModule, aszResource, 0);
        break;
    case DRV_FREE:
        /* free the custom command table */
        mciFreeCommandResource(wTableEntry);
        break;
}
```

Figure 10: Registering the Custom Command Tables with MCI in *DriverProc()*.

3.1.1.6 Using the New Command Set

Now that we have our MCI interface defined and working, lets go through an example of how to integrate the newly created MCI interface into Windows. We have defined both an ASCII command-string interface and a message-based command-message interface for motion control. Now let's proceed through an example to illustrate how the MCI parser would interpret the string: ***jog pmac0 axis 1 speed 5.5 notify***. The parser reads the string to determine what device driver the request is going to and the actual command to carry out. The parser uses the device name *pmac0* to map it into the destination device id (*dwDriverID* - Figure 9) for ***DriverProc()*** to use. In addition, the parser uses this information to determine what custom command table should be used to continue parsing the current command-string. In our example, the *pmac0* alias is used to tell the parser to use the ***pmac*** custom command table that we have defined in Figure 8. The zero suffix is appended to distinguish between multiple device driver instantiations. Several locations are searched when the parser looks for the command tables associated with the *pmac0* device alias. In order of priority they include:

1. any command table contained in external files for run-time linking,
2. all device-specific (new custom command tables) tables bound during link-time,
3. all device-type tables such as those for videodisks,
4. the core command set table for play, stop, etc.

The delimiter MCI_COMMAND_HEAD in the command table is used as a sentinel for the parser to begin. When that identifier is reached the command is compared with the string ("**jog\0**") location in the first column. If a match is found, the DWORD in the second column (MCI_SERVO_JOG) is used as the message field parameter to the multimedia driver. Notice that the MCI_SERVO_JOG command is the custom message used by the MCI_SERVO_JOG_PARMS data structure. As the parser recognizes each string it processes, it essentially fills in the command-message data structure with the appropriate commands and flags. Upon completion of the parsing phase, the newly constructed data structure is sent off to be processed by the ***DriverProc()*** entry point in exactly the same fashion as if a command-message using ***mciSendCommand()*** was issued. Continuing with the example, the parser tries to match each subsequent string element. For example, the next element is ***axis*** and has an MCI_INTEGER descriptor in the third column. This indicates to the parser that additional data is available from the command string. The parser then reads in the next string of data associated with MCI_INTEGER, converts it to an integer and stores it in the DWORD *axisInt* of the data structure corresponding to the MCI_SERVO_AXIS. This parsing process continues until the end of the string is reached. The order of parsing is irrelevant, because the parser maps the strings found to command messages and data structures. Therefore, if we had entered the same string in a different way such as ***jog pmac0 notify speed 5.5 axis 1***; the parser would generate the identical sequence of messages for MCI.

Finally, we discuss how the command-message interface could be integrated and used in a Windows-based application. The example in Figure 11 illustrates a typical use of the MCI_SERVO_JOG_PARMS data structure — the data structure used when a ***jog*** command is sent. This example comes from the MCITest program (see Section 4.5) we developed to exercise the MCI interface.

```
void Jog(HWND hwndDlg) {
   MCI_SERVO_JOG_PARMS  mciJogParms;
   DWORD dwStatus;

        mciJogParms.axisInt = 1L;
        dwStatus = mciSendCommand(mciID, MCI_SERVO_JOG, (MCI_SERVO_AXIS |
                                     MCI_SERVO_POS | MCI_NOTIFY),
                                     (DWORD)(LPVOID)&mciJogParms );
}
```

Figure 11: Sending a MCI-based MCI_SERVO_JOG Message.

The ***Jog()*** wrapper function simply allocates a structure of type MCI_SERVO_JOG_PARMS. The motor axis number (1L) we want to jog is hardcoded here for simplicity. The arguments to the ***mciSendCommand()*** are as follows: the *mciID* is a global device identifier that was returned when the device was first opened. The MCI_SERVO_JOG parameter is the MCI command message, and the flags *OR'd* together include the axis, move in positive direction, and notify the application when the command completed. The last parameter is the pointer to the structure MCI_SERVO_JOG_PARMS. Upon return from ***mciSendCommand()***, a return code in *dwStatus* can be checked allowing a string associated with the result to be printed using the ***mciGetErrorString()*** call. In the Windows application MCITest, the ***Jog()*** function was typically invoked as the result of pressing a Jog button on the user interface. Section 4.5 describes in further detail the MCITest application we used to test the MCI driver interface.

3.2 Low-Level Application Programming Interface For Motion Control

In addition to the high-level MCI interface, the NIST multimedia driver architecture provides a low-level interface as well. The low-level interface API (i.e., the ***motionJogStop()*** function shown in Figure 12) was implemented as a Windows multimedia compliant interface to provide:

a) additional performance related capabilities,
b) access to motion control board capabilities otherwise inaccessible through MCI,
c) a vehicle for sending and receiving large amounts of data, and
d) as an alternative multimedia interface for testing purposes.

The functions included in the NIST low-level API would be exported via the multimedia driver to applications. Instead of sending messages via the MCI interface, the software developer would use the API calls directly in the application software. Providing a low-level function-call based API in addition to the MCI capability is typical of most peripheral-oriented multimedia device drivers. A fully developed low-level API is seen as being critical to implement future motion control-based capabilities. Many of the issues regarding performance bottlenecks in the driver and tapping the additional capabilities of motion control boards will need to be addressed using this low-level API.

3.2.1 An Example API Implementation

The implementation details of the ***motionJogStop()*** function call encapsulate in many respects many of the similarities found with any follow-on low-level API calls a developer would implement. Figure 12 illustrates the code snapshot implementing the actual***motionJogStop()*** low-level API call found in the device driver code. Although specific to the Jog command, the functional structure is generic and typical of how basic command and/or data transfer routines would be accomplished.

```
VOID motionJogStop(void) {
  CHAR buffer[80];
  SENDBUFFER SendBuffer;

  wsprintfA(buffer, aszJogFormat, 1, aszServoJogStop);
  SendBuffer.lpData = (LPBYTE)buffer;
  SendBuffer.dwBufferLength = sizeof(buffer);
  MC_SendCommand (pBoardInfo->vh, (LPSENDBUFFER) &SendBuffer);
}
```

Figure 12: The *motionJogStop()* low-level Multimedia API Call.

Briefly, this function's responsibility is simply to stop a specified motor axis from being jogged (rotated). The function allocates a data structure that will provide storage for the jog stop ASCII command string (***aszServoJogStop***). *SendBuffer* is then assigned the pointer to the ASCII command buffer and the size of the string. The ***MC_SendCommand()***[11] function is then called with the pointer to the buffer and a pointer to a global handle representing the device driver. The ***MC_SendCommand()*** function simply calls the user-mode routine that sends the request down to the kernel-mode driver. This function is described further in Section 4.3.2 and characterizes the important linkage between the user-mode and kernel-mode drivers in this architecture.

[11] All user and kernel-mode specific device driver functions begin with the prefix MC, meaning motion control.

4. Kernel-Mode Device Driver Design

The user-mode multimedia device driver previously discussed provides motion control-based application software with the device independent methods for accessing the capabilities of the PMAC motion control board. The Windows-based multimedia device driver is contained within the Win32 subsystem and therefore must execute in user-mode. In Windows NT, user-mode applications or subsystems cannot access the hardware directly; therefore, they must have a paired kernel-mode native Windows NT device driver to facilitate all hardware access. The NIST paired device driver strategy thus provides applications with multimedia-based device-independent access to the PMAC board functions through a device-dependent in-kernel Windows NT device driver. In general, the kernel-mode device driver provides all hardware-level access to the PMAC on behalf of the multimedia driver. In high-level terms, the Windows NT kernel-mode driver can be divided into two areas of functionality; namely:

1. Required functions supporting the Windows NT model for device driver construction, and

2. Hardware specific functions required to interact with the PMAC motion control board.

Describing the implementation and use of any required low-level device driver functions would be meaningless without first providing a reasonable background of the underlying operating system's device driver data flow and system interaction models. This background is provided to the reader in the following section. Later sections describe the PMAC specific functions that were needed to control the motion control board at the kernel level in the Windows NT environment.

4.1 Windows NT Device Driver Model

In general, all operating systems provide an implicit or explicit model for controlling the flow of data to and from its hardware resources, with the ultimate goal of the device driver to provide application software with access to the hardware in the system. Software-based access to devices typically uses a file-oriented strategy for controlling the device; whereby, devices are only visible in the user-mode realm as named file objects that must be opened. This means application software interacts with devices (via device drivers) through operations typically used to do normal I/O with a file in the operating system. More specifically, in these types of systems, application software has the ability to open, close, read, write, or control various device characteristics with the file. Windows NT also provides this file-oriented view of device drivers by using what are known as file objects in the NT system. Windows NT provides this object-oriented file view of device driver interactions by implementing what is known as the Windows NT executive I/O model[12].

The Windows NT executive I/O model essentially combines a rigidly defined standardized interface for device driver communication with object-oriented concepts for representing devices. The model encapsulates these concepts with an abstraction called an I/O Manager. Figure 13 illustrates the location of the I/O manager in the Windows NT system. Notice how device drivers

[12] The Executive is the collection of components that form the base NT operating system. Executive components include: executive support, kernel, memory manager, cache manager, process structure, interprocess communication (LPC and RPC), object manager, I/O manager, configuration manager, hardware abstraction layer, and the security reference monitor.

are enveloped by the I/O manager, implying that specific guidelines must be adhered to in order to carry out meaningful interactions with the device driver.

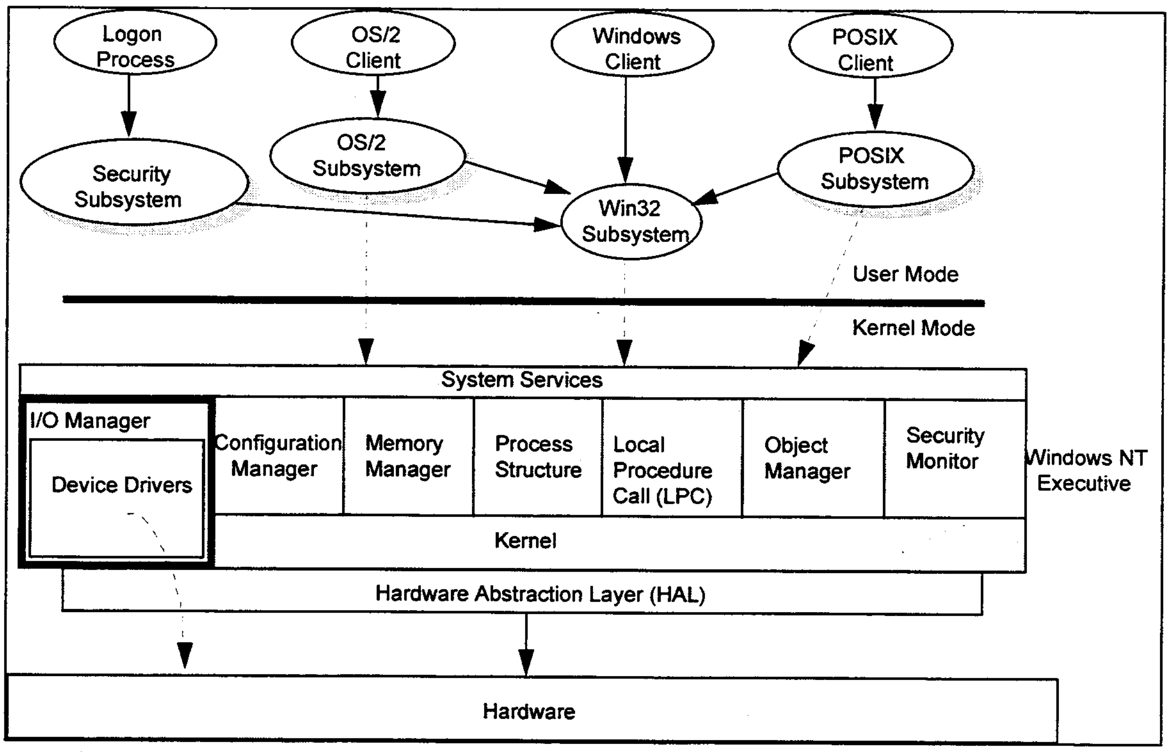

Figure 13: The I/O Manager Component in the Windows NT System.

The I/O manager component provides a consistent, object-based, and packet-driven I/O style of device driver interaction. The primary function of the I/O manager is to provide a means to accept I/O requests, create a standardized packet of information to represent the request, and to route the request packet to the appropriate device driver. In addition, the I/O manager must monitor the request packets completion status, and finally return this status to the calling program. In order to accept I/O requests in the first place, the I/O manager defines and exports a single I/O model with a set of standardized kernel-mode support routines to manipulate NT device objects. Consistent interfaces between the originator of the request and the device driver that must handle the request are supplied by the I/O manager. Any native Windows NT application or subsystem can make requests on device drivers using the aforementioned file abstraction. Win32 applications, display programs, or printer drivers all request kernel-mode driver services via standardized Win32 subsystem API services (i.e., ***CreateFile(), DeviceIoControl(), etc.***). The Win32 API call to open or control the device/file is mapped onto a specific I/O manager area in the system services part of the NT executive (see Figure 13).

The I/O manager is responsible for exporting the I/O specific system services initially; therefore, all requests for file/device interaction will be received by the I/O manager. The request is then packetized by the I/O manager in a form suitable for consumption by the kernel-mode driver. The basic unit or packet of information used to build a request in the I/O manager is called an Input/Output Request Packet, or IRP. This is the basic I/O manager structure used to communicate with kernel-mode device drivers in the NT system. An IRP packet consists of two parts: a header, or fixed part of the packet, and I/O stack locations. The header is used by the I/O manager to store parameters (i.e., open device driver command) and the address of the device object (i.e., file handle) from the original request (callers parameters). The fixed part is also used during the return from the call, where the device driver places the status of the requested operation in the header part. The I/O manager uses the I/O stack location part to set device driver specific parameters. For example, the particular operation requested is transferred by the I/O manager into the I/O stack location as a special function code understood by the driver. The driver then uses that special function code in the driver to dispatch a particular function to handle the request. The I/O manager defines a set of standard routines for NT driver implementers to manipulate IRP's, thereby providing a consistent interface for all kernel-mode drivers. The I/O manager supports NT driver development using a single I/O model, a set of kernel-mode routines to execute I/O operations, and a consistent interface between the requester and the NT driver that fields the request. The required I/O manager elements needed to begin developing the framework for the NIST kernel-mode Windows NT device driver are discussed in the next section.

4.2 Elements of a Windows NT Device Driver

During a Windows NT kernel-mode device driver development effort, a skeleton driver is typically created using the many inherent Windows NT provided functions, routines, and macros. Adherence to this rigid driver structure used by Windows NT provides a consistent interface for developing device drivers. The consistency of the interface allows other device drivers to communicate with each other in an abstract manner (object-based) while providing cross-hardware support for a variety of microprocessor architectures (MIPS, INTEL, PowerPC). These important functions used during the driver development process form the basic skeleton of a minimal driver which can then be compiled and linked into the Windows NT system. Once the shell has been properly integrated, the developer then begins to add the device-specific functionality required for the hardware being used. Depending on the type of hardware device being used, the device driver may require a great deal of internal NT support functions or only a small subset. The PMAC board essentially required various methods for communicating commands, manipulating memory mapped RAM, and receiving interrupts from the board. Therefore the requirements of the driver were such that we did not have to provision the driver with every possible inherent Windows NT driver function available. In support of a generic NT device driver framework, we included the following standard NT defined functions:

- I/O manager defined functions for driver Initialization, Interrupt Servicing, and IRP Queuing.
- Ancillary functions to allocate spinlocks and interrupt objects, manage memory, use the registry, and setup symbolic links to the physical device name.

In order to support the device-specific portions of the device driver, many other Windows NT Executive defined functions were needed. These functions are explained later in the section on PMAC hardware integration functions (Section 4.3). The Windows NT specific portion of the device driver is reasonably generic and similar in scope to most device drivers written for the NT environment. These functions will be briefly summarized. For a more in-depth discussion of each entity and those not discussed here, developers should consult with the documentation and source code examples available from Microsoft in the Device Driver Development Kit (DDK) for additional information.

4.2.1 NT Device Driver Initialization

Depending on the level of the device driver or the nature of the underlying physical device, a Windows NT kernel-mode driver must provide several I/O manager defined routines[13]. In a library-based system such as Windows NT, the transfer address of the device driver entry point must be linked into the OS loader (the boot stage Windows NT program that loads the kernel image into main memory) to allow initialization. The name of this transfer address in driver space is the ***DriverEntry()*** initialization routine. By naming the entry point ***DriverEntry()*** this linkage is automatically setup; otherwise, the developer would have to specify the new entry point name to the linker. This is the first function Windows NT calls when initializing the driver. Upon initialization, the ***DriverEntry()*** routine is responsible for exporting additional I/O manager defined entry points, initializing the Windows NT device object the driver will use, and for setting up, or requesting any system resources needed.

Prior to any application issuing requests to the device driver, the ***DriverEntry()*** function creates a special device object that is bound to the underlying physical device. This binding or association phase commences when a call to ***IoCreateDevice()*** is made. The ***IoCreateDevice()*** function allocates space for the device object. A pointer is created as a result of a successful return from this function that provides a handle the device driver will use to access the hardware specific portions of the device. After the device object has been created and bound, several other staged initialization functions typically begin as part of the ***DriverEntry()*** routine; namely:

- Set up and export ***Dispatch()*** function entry points.
- Interrogate the device or the NT Registry to gather device configuration information.
- Report hardware resources needed to NT and check for conflicts.
- Initialize the physical device.
- Connect and Test the Interrupt Service Routine.
- Signal the User-mode Driver that the Kernel-mode driver has loaded successfully.

An important role of the ***DriverEntry()*** function is to assign addresses to the ***Dispatch()*** and ***Unload()*** functions. After the device driver has been initialized and is placed in an operational state, it can only handle request for services that the driver's implementers supplied. In addition to custom capabilities that the driver implementers may provide, most NT drivers should be able to handle create, close, read, write, or device control requests[14]. The ***Dispatch()*** routine is the

---

13 Windows NT kernel-mode drivers are divided into three specific types lowest-level drivers, intermediate drivers, and file system drivers. Lowest-level (also called device drivers) directly control a physical device. The NIST NT kernel-mode driver is a lowest-level device driver. Intermediate drivers represent device-specific classes of drivers such as a virtual disk driver abstracts a real hard disk device driver. Finally, a file system driver supports the system supplied FAT, HPFS, CDFS, and NTFS file systems in the NT machine.

14 In the NIST kernel-mode device driver, we did not implement READ, WRITE, or custom major function code ***Dispatch()*** capabilities. I/O with the motion control board was facilitated using device control requests for character-based and memory-mapped data manipulation exclusively. READ/WRITE requests are typically used during large DMA-based data transfers. The PMAC board does not support DMA-based data transfer.

centralized function that receives IRP's routed from the I/O manager and dispatches functions to handle such requests. It is the responsibility of the ***DriverEntry()*** routine to have these routines exported during the driver's operation to correctly field these requests from the I/O manager. By providing these addresses, the driver effectively exports to the I/O manager the entry pointers to the driver for all possible requests handled by the driver. Figure 14 illustrates how the assigning of function addresses for the ***Dispatch()*** and ***Unload()*** routines are handled in the ***DriverEntry()*** routine.

```
pDriverObject->DriverUnload                               = MC_Cleanup;
pDriverObject->MajorFunction[IRP_MJ_CREATE]               = MC_Dispatch;
pDriverObject->MajorFunction[IRP_MJ_CLOSE]                = MC_Dispatch;
pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]       = MC_Dispatch;
```

Figure 14: Setting up the *Dispatch()* Table in Windows NT.

The ***MC_Cleanup()*** function has been defined as the device driver's ***Unload()*** function address. This function is called when the device driver is taken out of memory by the NT system. The ***MC_Cleanup()*** function simply unmaps all memory used, disconnects the interrupt interface, and reclaims all resources used by the driver back to the NT system. Notice that all the major function codes for creating, closing, and device control have all been set to the same function address— ***MC_Dispatch()***. This process centralizes all requests to the driver in just one functional area; however, this approach is not mandatory. Indeed, there could be a single dispatch function for each major function code handled by the driver. The centralized approach is typically opted for in many designs if the number of major function codes is not significantly high.

The abbreviated ***MC_Dispatch()*** routine is shown in Figure 15. The functionality normally associated with each case statement identifier in Figure 15 have been omitted for clarity and will be addressed in a later section.

```
NTSTATUS  MC_Dispatch(
   IN PDEVICE_OBJECT pDeviceObject,
   IN PIRP pIrp)
{
   PIO_STACK_LOCATION pIoStack;
   NTSTATUS Status;
   PDEVICE_INFO pDevInfo;
   ULONG IoCode;

   switch(pIoStack->MajorFunction) {
         case IRP_MJ_CREATE:
         case IRP_MJ_CLOSE:
         case IRP_MJ_DEVICE_CONTROL:
               IoCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;
               switch(IoCode) {
                     case IOCTL_MOTION_READ_PORT:
                     case IOCTL_MOTION_WRITE_PORT:
                     case IOCTL_MOTION_GET_DPRAM:
                     case IOCTL_MOTION_SEND_COMMAND:
                     case IOCTL_MOTION_INTR_INIT:
                     case IOCTL_MOTION_INTR_TERM:
   }
```

Figure 15: An Abbreviated Windows NT Kernel-Mode Driver Dispatch Function.

Figure 15 illustrates where the connection between the NT device driver and the developers placement of the code necessary to carry out the drivers hardware-specific capabilities can be found. For example, all PMAC hardware related methods implemented, such as mapping Dual-

Ported RAM into NT space, performing raw hardware port I/O, and initializing the interrupt interface were all done using special device control codes utilized in the ***MC_Dispatch()*** routine. Use of special device codes, or Input/Output Control codes (IOCTL's) are typically used by many operating system designers to designate the functional entry points to the kernel device driver interface.

In the centralized function ***MC_Dispatch()*** shown in Figure 15, a case statement determines what major function code has been received by inspecting the I/O stack location of the IRP. If the major function code is of type IRP_MJ_DEVICE_CONTROL, then the specific IOCTL code is determined by parsing the I/O stack parameter location. After the type of IOCTL has been determined, the function responsible for carrying out the necessary IOCTL is called (not shown in Figure 15). This provides a central hub for all incoming device control requests.

After the ***Dispatch()*** entry points for the driver have been defined, the device-specific configuration information such as the hardware I/O port range, interrupt requested, the base address of memory mapped RAM is obtained by probing the device directly if the hardware supports this, or by querying the Windows NT Registry. The information returned from the Registry/device is used to request resources from NT and to detect resource conflicts. Reporting resources and detecting conflicts within Windows NT drivers is provided by the kernel call ***IoReportResourceUsage()*** function. If the resources can be claimed by the driver, then they are written to the Registry as part of the driver load function.

At this point, the actual physical hardware can be initialized. Hardware initialization is a generic function needed in all device drivers. However, because devices differ in great respects, very specific methods to bring each on-line and in stable state are required. The PMAC initialization occurs as part of the ***DriverEntry()*** function. The PMAC's initialization procedure requires writing various hexadecimal codes to the onboard registers of the PMAC at various offsets to the base I/O address. Each base offset provides access to specific data areas such as the PMAC data registers for sending and receiving information or the interrupt control word for acknowledging interrupts.

To properly initialize the PMAC, BUS communications must first be brought on-line providing command-driven capabilities to applications using the BUS. Initialization procedures writing or reading to the base I/O address offsets of the PMAC are all done using wrapper functions we developed to front-end Windows NT macros that provide raw hardware I/O port addressing (see Section 4.3.2 and the ***MC_In()*** routine). Values are sent to all data registers of the PMAC to clear them prior to usage just for safety and is recommended. Hardware initialization begins by enabling the Programmable Interrupt Controller (PIC). There are two PIC's in the Intel-based system that we are using. The on-board PMAC PIC is an Intel i8259A controller that has the ability to generate eight different types of interrupts to the PC. In the old DOS/Windows version of the device driver, separate code to write values to the PC's PIC were also used; however, this should not be done in the Windows NT environment. The PC's PIC will be initialized to the correct values during the NT kernel call that actually connects the Interrupt Service Routine we developed to the NT system. This will occur later after the interrupt mask has been set prior to connecting the interrupt to NT. Depending on the type of interrupts the driver wants to be made aware of when they occur, a mask can be applied to the PIC to filter those particular interrupts only. This hardware method for selectively masking interrupts is provided all the way up to the application level. That is, as part of the original application scenario, a user has the ability to request what types of interrupts will be masked, and the driver will respond by masking off only those specified. We maintained this capability in the current driver to provide backward compatibility. However, as part of the initial board setup, all interrupts are masked off until the actual interrupt service routine is connected with the NT system.

Interrupt testing routines in the driver were only concerned with testing whether or not interrupts were actually being generated on the PMAC. A for-loop designed to force a specific interrupt every 1/25 of a second was entered. If interrupts were occurring at a substantial rate then the testing phase returned a successful status and the ISR was then actually connected to the NT system. When successful initialization of the driver has completed, the ***DriverEntry()*** routine finally returns a successful load status to the user-mode device driver by writing special values to the Registry that the user-mode driver reads and validates when it starts.

4.2.2 NT Specific Interrupt Service Requirements

If a hardware device such as the PMAC generates interrupts, then the device driver must include an Interrupt Service Routine (ISR) function. The internals of the ISR are quite specific to the PMAC device and thus are discussed in detail in Section 4.3.1 on the PMAC specific NT driver functions. However, the structured process for developing, installing, and connecting to an interrupt using an ISR contains Windows NT specific functions as defined by the kernel. There are generic ISR functions available through NT that bind the ISR routine to the NT hosted system. Figure 16 illustrates the driver function ***MC_ConnectInterrupt()*** that contains the NT specific functions to connect the ISR to the NT system.

```
BOOLEAN MC_ConnectInterrupt(
  PDEVICE_INFO pDevInfo,
  ULONG Interrupt,
  BOOLEAN bLatched)
{
  KAFFINITY Affinity;
  KIRQL InterruptRQL;
  ULONG Vector;
  NTSTATUS Status;

  dprintf2(("connecting to interrupt 0x%x", Interrupt));

  Vector = HalGetInterruptVector(pDevInfo->BusType, pDevInfo->BusNumber, Interrupt,
                                 Interrupt, &InterruptRQL, &Affinity);

  Status = IoConnectInterrupt(&pDevInfo->InterruptObject, MC_InterruptService,
                              pDevInfo, NULL,Vector, InterruptRQL,InterruptRQL,
                              bLatched ? Latched : LevelSensitive, FALSE, Affinity, FALSE);

  return ((BOOLEAN) NT_SUCCESS(Status));

}
```

Figure 16: Connecting the Interrupt Service Routine with Windows NT.

In this function, several key procedures need to take place in order to connect the ISR to NT. First, the kernel call to ***HalGetInterruptVector()*** uses the Hardware Abstraction Layer (HAL) of NT to return a portable representation of the interrupt to use with NT[15]. Given a specific choice for

[15] HAL - Hardware Abstraction Layer, refers to the only non-portable component of a Windows NT Executive (kernel). The HAL exports routines that abstract hardware platform-specific details about the hardware ports, buses,

an interrupt vector number, i.e. 10 decimal, the HAL layer does the mapping needed to represent the chosen interrupt vector into a machine independent fashion. This and other similar hardware layer abstraction techniques allow NT to provide cross-platform support for device driver code. The second aspect of this function actually uses the number returned by HAL to connect the named ISR to the system using the ***IoConnectInterrupt()*** NT kernel call. Finally, the function must provide an address of the user developed ISR function to map into NT when an interrupt occurs. The user supplied function we developed is called ***MC_InterruptService()***. This function essentially is an interrupt acknowledgment routine that queues another procedure called a Deferred Procedure Call (DPC) to carry out most of the work associated with the interrupt event.

A DPC is queued to perform the actual functions of the interrupt at a later (lower interrupt dispatch priority) time. This concept of queuing work to be done at a later time is typical of most multi-tasking operating systems and provides a fundamentally different ISR model then previously used in the DOS/Windows environment. That is, under DOS/Windows setting, the entire microprocessor interrupt system is held up until the ISR terminates. In NT as well as many other multi-tasking operating systems, this behavior would be disastrous. This is a key area to consider when targeting device drivers from DOS/Windows to operating systems such as NT.

4.2.3 IRP Queue Management Routines

As part of the interrupt processing algorithms of the kernel-mode device driver, the need to initialize and maintain queue management functions for the IRP's arose. IRP queue processing includes providing textbook style functions capable of enqueing and dequeuing IRPs. The ability to peek inside a queue for a specified IRP was also provided although it was not used in this phase of testing.

Ancillary functions to provide synchronization (spinlocks and mutexes) were needed during access to the queue functions as contention for resources during ISR instances would require it. The ISR executes at a processor priority of DISPATCH_LEVEL in Windows NT which is the highest priority[16]. If mutual exclusion primitives are not used then data in the ISR as well as the queue could be overwritten as a minimum due to multiple accessing.

4.3 PMAC Specific Device Driver Functions

In addition to the NT specific areas that form the skeletal driver, the device-specific portions were developed and integrated to form the functioning PMAC component of the NT kernel-mode device driver. This area essentially provided the driver with the capability to communicate with the PMAC board using a variety of hardware-specific methods, as well as provide hardware initialization, testing, and shutdown facilities. Some portions of the NT specific area (i.e., connecting the Interrupt Service Routine to NT) although inherent to NT required PMAC device-specific support in the driver and therefore are discussed in detail.

---

and interrupts. Moving the Windows NT kernel from one platform to the next simply requires the developers of Windows NT to rewrite the HAL for the new target platform. This fact has put Windows NT into the realm of being as portable as UNIX to different microprocessors.

[16] Interrupt Request Levels within NT are broken down into several different types that specify differing priority levels for hardware interrupts. DISPATCH_LEVEL is the highest priority task; the ISR executes at this level.

By far the most critical implementation detail of the kernel-mode device driver was providing the communication pathways from the application via device drivers to the motion control board. There are a variety of generic ways to provide the necessary communication paths. In the case of the PMAC, it provides communications using serial ports, BUS-based (parallel I/O Port address), Dual-Ported RAM, and Interrupt-driven paths. The Dual-Port RAM ability is an option so it may not be present in some systems (although unlikely). In the next sections we will discuss how we provided Interrupt, Dual-Port Ram, and BUS-based communication in the device driver. Serial port communications are provided by the PMAC; however, we did not implement this as serial drivers are readily available and can be integrated rather easily if the need arose. In our testing we did not need to provide this functionality.

4.3.1 Interrupt Services for PMAC Communication

The Interrupt Service Routine and the Deferred Procedure Call support routines contain a great deal of PMAC specific code. In general, the ISR and DPC together provide the Windows NT capability to service interrupts from the PMAC. Unlike DOS/Windows, when an interrupt occurs the entire machine is not disabled from interrupts while the interrupt is being serviced. NT is multi-tasking; therefore, stopping the machine from interrupting the CPU would be disastrous for any length of time. NT provides a model of servicing interrupts by acknowledging the interrupt on occurrence and then queuing a DPC routine for a lower interrupt request time to do the actual work of the ISR. In fact, you will get much better interrupt servicing response time in a system such as NT if the ISR in general does *not* perform any time consuming tasks. The DPC routine is the actual ISR function that must handle the interrupt. In this case, the DPC in the NIST NT driver is the workhorse of the entire interrupt process. The DPC mainly allows the driver to extract the type of interrupt that occurred and any data associated with that interrupt for later notification to the application. In order to discuss the interrupt process at this level, it becomes necessary to provide some background into how the original applications used interrupts in the DOS/Windows setting and what they do with them. We then can discuss the user-mode interaction with the kernel-mode driver to facilitate the communication of interrupts from the PMAC board to the kernel-mode driver and ultimately to the application.

4.3.1.1 DOS/Windows Interrupt Architecture

In the DOS/Windows setting, fielding an interrupt from an Intel-based machine proceeds in the following manner: when an interrupt occurs, the application's ISR would call an Intel-specific C run-time function called ***disable()*** to actually disable all interrupts on the Intel i8259A PIC of the AT-based machine[17]. All interrupts on the PIC of the PMAC motion control board would also be disabled at this time. The ISR would then acknowledge receipt of the interrupt from the PMAC PIC. The application's ISR would then set a flag and post a message to the application's message-loop, telling of an impending interrupt. The ISR would then re-acknowledge the PMAC interrupt, unmask the appropriate interrupts, and finally, call ***enable()*** to allow interrupt generation on the AT-based machine's PIC. During that point when the message is posted, the application responds to the interrupt message by doing whatever post-processing is required. At this point the machine continues on with whatever it was doing. Clearly, this happens in a very short amount of time; however, it becomes evident that this method of enabling/disabling interrupts on the CPU to get

[17] Disabling the AT machines' PIC under Windows NT is not possible. After the ISR is connected to the interrupt vector then the ISR must determine if the interrupt is in fact for this ISR or it is spurious. In a multitasking OS like NT, disabling all interrupt generation on the machine would be disastrous.

ISR work done when migrated to a multitasking and multi-user operating systems such as Windows NT is unreasonable.

The interrupt capabilities in the Windows NT device driver structure emulate many of the functional methods used by the original DOS/Windows interrupt service handler. We wanted to not only implement the ISR using inherent Windows NT functions, but also emulate the personality (the application interaction with the ISR) of the application-interrupt capability from DOS/Windows. By doing this, we could measure performance characteristics and recommend alternative strategies for bridging the DOS/Windows style of interrupt interaction with that of an advanced graphical OS environment such as Windows NT.

4.3.1.2 NT Interrupt Processing Overview

The interrupt handling capability of the NIST device driver architecture is embedded throughout all three layers of the architecture. That is, interrupts initially are generated on the PMAC and are received in the kernel-mode ISR routine. As part of the kernel-mode ISR implementation, the user-mode multimedia driver is notified of this event. The user-mode driver then notifies the application's messaging loop that an interrupt has occurred via a callback or messaging mechanism. In order for the application to receive this *pseudo* interrupt from the PMAC, it must do so via several layers of device and application interaction. Clearly, this interrupt path involves significant latency overheads. The next section on performance discusses why this particular architecture was chosen and some of the issues it raises. Figure 17 illustrates the interrupt processing algorithm we used to support interrupt generation from the PMAC.

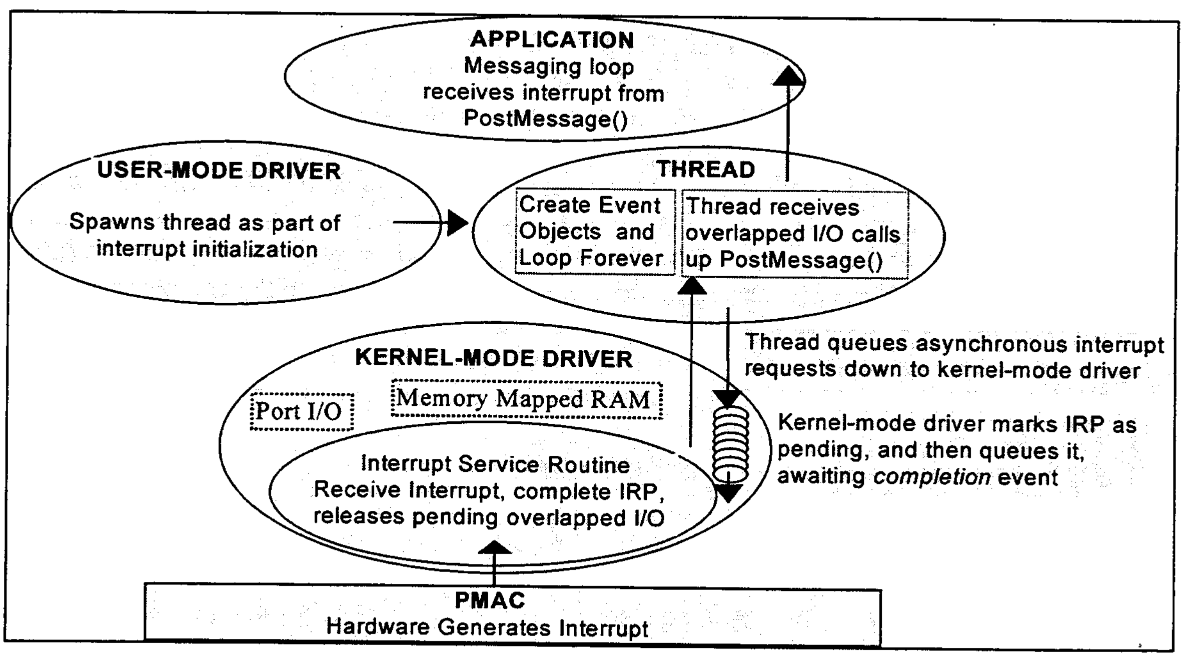

Figure 17: Interrupt Processing Algorithm in the Windows NT Drivers.

In general, the interrupt processing through the various device driver and application layers is accomplished using the following algorithm: the user-mode driver spawns a thread, creating an event-based asynchronous request/signal feedback loop. A forever loop waits on the possible events from several event objects created pertaining to any possible interrupt that can occur.

These events are queued down to the kernel-mode driver (in the form of IRP's) as asynchronous request for pending overlapped I/O completion.

The kernel-mode driver cannot complete the request until an interrupt happens from PMAC; therefore, it must use the queue management routines from Section 4.2.3 to queue the request as an IRP on its interrupt queue. Upon receipt of an interrupt from PMAC, the kernel-mode driver will: acknowledge and prevent PMAC from continued interruption, determine what type of interrupt occurred, unless it is spurious (not meant for this ISR), dequeue any IRP from the interrupt queue, fill-in the necessary return buffer with the type of interrupt and finally, *complete* the IRP. *Completing* the IRP in the kernel-mode driver immediately signals the event object in the user-mode driver by releasing the pending overlapped I/O event flag. The thread in the user-mode driver catches the event and sends a message back to the application that an interrupt has occurred. The application then receives the interrupt in a special messaging loop (message pump) that has been setup to receive all incoming interrupt messages from the driver. The handle or instance data for the window or callback has been passed to the driver from the application in an initial call to MCI through the user-mode driver setup routine that we defined. The application decodes the parameters of the *EvPmacNotify*[18] message upon receipt and uses the decoded information (i.e., what interrupt type) to perform some post-processing. After returning from the message send routine, the user-mode driver thread immediately queues another interrupt request down to the kernel-mode driver asynchronously, and continues to wait for other interrupt event objects to be released.

In the Windows environment, the latency involved with getting the message back to the application (through the Windows messaging pump) has been the major bottleneck. Several interfaces for transmitting the message from the user-mode driver to the application have been exercised to find ways of minimizing the message communication response time. Some of the interfaces include ***DriverCallback()***, ***SendMessage()***, ***PostMessage()***, and ***mciDriverNotify()***. The ***DriverCallback()*** interface sends a message back to an application via either a callback function or to a window handle. The ***DriverCallback()*** function is the Multimedia-based method for doing this sort of communication. ***SendMessage()*** is the Win32 API which sends a message to an application and does not return until the message has been received. ***PostMessage()***, another Win32 API, posts a message to an application's message loop and returns immediately to the calling program. The ***mciDriverNotify()*** API has also been used for testing purposes only. It is inherently part of the MCI driver interface and presents only four possible messages. Because this API is part of the internal multimedia structure of Windows, it would not be prudent to extend this interface.

4.3.1.3 Interrupt Algorithm Performance Issues

In general, interrupt-driven communication approaches are the most efficient; however, they are much more difficult to implement with respect to hardware. Under the umbrella of the DOS/Windows implementation of interrupt-driven I/O this is certainly the case. However, as we found out in our Windows NT implementation of interrupt-driven I/O, this was not the case. Most of the issues precluding efficient implementations of the interrupt-driven approach resulted from the need to provide application-level support for interrupt notification. Whenever the higher level Windows subsystem (ring 3) interacts with the lower-level (ring 0) kernel, it causes time

[18] *EvPmacNotify* refers to the message currently used in the Delta Tau Windows 3.1 Software Release. When interrupts are initialized on the PMAC, a special message-loop executes waiting for this message to arrive. After arrival, the message parameters are then decoded and used to determine what type of interrupt has occurred.

consuming context switches which preclude efficient interrupt notification algorithms. For notification situations in which the latency of application-interrupt receipt can tolerate times in the 100ms range, this algorithm is very well suited. In addition, the algorithm is sufficiently generalized to be extended for data transfer cases as well. That is, if a buffer transaction scheme were needed, this algorithm could be generalized enough to provide these capabilities and integrated quickly. However, to remain backward compatible with the application base and for testing capabilities this particular algorithm remains an optimal choice for these purposes[19]. It is simply not advisable however to service interrupts all the way up to the application *unless* the interrupts are occurring at such intervals that the application and kernel-mode driver can tolerate them; otherwise, the kernel-mode driver (located at ring 0) will simply overrun the application due to the large latency inherent in the Windows NT message routing and communication subsystem (located at ring 3).

Several interrupt algorithms have been tried to minimize the response time and to provide reasonable event processing during *interrupt storms*[20]; including: **(1)** queue as many interrupt request to the driver for all possible interrupts (eight in the PMAC), **(2)** create an arbitrary threshold before we begin to queue more interrupts, **(3)** build in logic to determine when to send interrupt requests, and **(4)** use multiple threads with critical sections for background data management tasks.

We also modified operating system scheduling algorithms that would minimize the interrupt latency problems at the hardware level, including modifying the processes class priority as well as the thread scheduling priorities for the real-time class and priority. In the kernel driver we tried to combine the ISR acknowledge and interrupt determination process into the ISR directly — thereby bypassing the queuing of deferred procedure calls to handle the interrupt response. We theorized that if we could reduce the response and fielding time associated with the interrupt determination by not using the DPC mechanism, then we could reduce substantially the response time. However, this would adversely effect the applications ability to field the interrupts. That is, the more speed-up of the kernel-mode interrupt fielding capability, the more the application/user-mode driver suffers in trying to keep up with it. In general, the ability of NT to accept and acknowledge interrupts in a timely manner is not a problem. In fact, we were profiling the driver at one point to determine how fast NT could react to an interrupt. It was empirically found in our environment that NT was responding to interrupts at the rate of around 10μs.

Interrupt latency measurements from within the kernel driver and the user-mode driver to application have been done. Currently, it takes approximately 160 μs to completely service an interrupt in the kernel-mode (using IRP queue method) driver. This includes the average time the kernel-mode driver takes to queue a DPC routine (i.e., the profiling takes place in the DPC routine so we have normalized this into the result). One debug print statement in the ISR causes a 10 ms delay in the total response, adding to the approximate 180 to 220 ms total round trip delay, and actually causing some interrupts to be skipped. Therefore, the round-trip message delay using either MCI or the direct API calls were profiled between 180 and 220 ms. This confers with other

19 Because of its inherent OS structure, Windows NT like many other multitasking operating systems is not particularly well suited to handle application-based interrupt notification. For this reason several algorithms have been proposed to emulate the DOS/Windows style of interrupt interaction. Through our research we have determined that the algorithm we have developed is optimal for the general case.

20 Interrupt storms occur when interrupts are generated by the hardware faster then they can be serviced by the ISR or application. The default procedure for dealing with these storms is to disable them.

colleagues stating that heavily loaded systems can average turnaround latencies of around 100 ms., one-way. Setting real-time threads and upgrading process classes to real-time did not significantly provide any better results, only to slow the GUI interface down on the NT host, as could be expected. Because of these intolerable latency delays, other alternatives and design issues surfaced during development.

4.3.1.4 Interrupt Design Alternatives

Limitations in the time it took for the messaging loop to react to the incoming interrupt callback from the user-mode driver forced the designers to form alternative approaches to the interrupt feedback loop problem. Because of the fundamental OS differences between DOS/Windows and Windows NT, the architectural issues of designing interrupt-driven device drivers need to change. The ideal option for developers is to migrate the interrupt and hardware intensive algorithms down as close to the kernel-mode driver as possible. Only allow the hardware to interact on an interrupt basis with the kernel-mode driver. Then in the user-mode driver or application, simply set up some sort of passive interrupt notification scenario; perhaps based on the overlapped I/O and event objects illustrated here. This way the direct coupling between the application and the interrupt capabilities have been severed. This will provide better serviceability of interrupts in the kernel-mode driver while maintaining reasonable application performance. This strategy was not done here because we needed to emulate the existing application requirements. If we were doing the application re-architecting in addition to the device driver, we would of opted for bypassing the interrupt notification process in the application altogether. In general, application-level interaction with interrupts is not a significantly robust method and needs to be redesigned before migrating into the Windows NT setting.

4.3.2 BUS Communications

BUS communications allow applications, via device drivers to send the PMAC ASCII formatted commands and to return result codes in ASCII string format. Formatted strings are sent through the hardware address I/O ports to the PMAC using BUS-based communication. This means that single bytes representing a command string are sent over the BUS using a specified I/O address port.

Before the hardware address for the I/O ports can be used from within the NT system for BUS communications however, a series of calls to the hardware abstraction layer of NT must again take place in order to map the hardware port I/O address in to Windows NT system space. Recall that the HAL layer provides a transparency layer to provide for cross-platform support for device drivers; therefore, a mapping must take place that translates the physical I/O address of the port (i.e., 0x210) into a comparable hardware address suitable to the underlying hardware.

Figure 18 illustrates the use of the HAL by mapping the hardware specific I/O address though NT using the I/O start address returned from the ***HalTranslateBusAddress()*** NT executive system call. Notice the actual I/O address shown in the call as ***PortAddress*** (0x210) has been mapped into an NT specific address based on what was returned by the hardware during the call. The newly mapped address may be very different from that of the one requested and is placed in the variable address named ***MappedAddress***. This new address is then set to the pointer ***pDevInfo->PortBase***, which will be used when the actual port accessing functions need a port address offset to read from or write to during the drivers execution.

```
/*
 * translate the bus-relative port address using HAL, map the port address into memory if necessary
 */
VOID MC_MapPort(PDEVICE_INFO pDevInfo, PUCHAR PortBase, ULONG NrOfPorts)
{
   PHYSICAL_ADDRESS PortAddress;
   PHYSICAL_ADDRESS MappedAddress;

   pDevInfo->PortMemType = 1;     // i/o space
   PortAddress.LowPart = (ULONG) PortBase;
   PortAddress.HighPart = 0;
   HalTranslateBusAddress(
         pDevInfo->BusType,
         pDevInfo->BusNumber,
         PortAddress,
         &pDevInfo->PortMemType,
         &MappedAddress);
   if (pDevInfo->PortMemType == 0) {
         /* memory mapped i/o - map the physical address into system space */
         pDevInfo->PortBase = MmMapIoSpace(MappedAddress, NrOfPorts, FALSE);
         dprintf2(("ports at 0x%x", pDevInfo->PortBase));
   } else { /* this is what we want to do for this driver */
         pDevInfo->PortBase = (PUCHAR) MappedAddress.LowPart;
         dprintf2(("%d ports at 0x%x (i/o space)", NrOfPorts, pDevInfo->PortBase));
   }
   pDevInfo->NrOfPorts = NrOfPorts;
}
```

Figure 18: Use of NT Hardware Abstraction Layer to Map Hardware Ports.

In order for the NT kernel-mode driver to provide the BUS communications capability using the newly mapped port I/O address, a set of NT specific hardware port access macros were used. Two methods shown in Figure 19 to access the port I/O routines from the device driver were provided. These two functions use the address (***pDevInfo->PortBase***) that was mapped into I/O space as the base address to use for port communication in the device driver.

```
/* output one BYTE to the port. bOffset is the offset from the port base address */
VOID  MC_Out(PDEVICE_INFO pDevInfo, BYTE bOffset, BYTE bData)
{
        WRITE_PORT_UCHAR( (PUCHAR) pDevInfo->PortBase + bOffset, bData);
}
/* input one byte from the port at bOffset offset from the port base address */
BYTE  MC_In(PDEVICE_INFO pDevInfo, BYTE bOffset)
{
        return (BYTE) READ_PORT_UCHAR( (PUCHAR) pDevInfo->PortBase + bOffset);
}
```

Figure 19: Wrapper functions to Access Hardware Ports in Windows NT.

During a read operation on the hardware port, the wrapper function ***MC_In()*** is called which simply uses a Windows NT specific macro (READ_PORT_UCHAR) to read a raw byte from the hardware port. This use of macros for reading hardware ports in Windows NT illustrates yet another aspect of how Windows NT device drivers remain portable across multiple classes of microprocessors. The device driver simply needs to read a character from a given hardware port using the hardware independent macro. The use of macros in this case forces the HAL layer of Windows NT to determine the underlying type of hardware it is using, (i.e., Intel, Mips, or PowerPC) before it reads the port. Because microprocessor architectures differ in the way ports are accessed (assembly language access), using this macro feature eliminates the device dependent differences found among them.

In the original DOS/Windows driver scheme, BUS communications used a polling paradigm to send each character of the command string down to the PMAC board one at a time via Intel specific C Language Runtime extensions called ***outb()*** and ***inp()***. Those familiar with developing device drivers for Intel-specific machines should quickly recognize these Intel instruction wrappers that send and receive, respectively, a single byte of data through an addressable I/O port to the PMAC[21].

```
/* Send a buffered command to PMAC via the Win32 DeviceIoControl() API call. */
BOOL    MC_SendCommand(MCUSER_HANDLE vh, LPSENDBUFFER lpSendBuffer)
{
   BOOL bOK;
   DWORD dwCount;
   PPMACSENDBUFFER pBuffer; /* driver can't use LPSENDBUFFER, can use PPMACSENDBUFFER */
   dprintf4(("pmac send buffer"));
   pBuffer = (PPMACSENDBUFFER) lpSendBuffer;
   bOK = DeviceIoControl(vh->hDriver, IOCTL_MOTION_SEND_COMMAND,
                         pBuffer, sizeof(PMACSENDBUFFER),
                         pBuffer, sizeof(PMACSENDBUFFER),
                         &dwCount, NULL);
   if (!bOK)  dprintf3(("SEND_COMMAND ioctl not completed"));
   dprintf4(("buffer sent to pmac"));
   return(bOK);
}
```

Figure 20: *MC_SendCommand()* Illustrates User to Kernel-mode Driver Access.

The method we developed in the NT driver to provide this polling sequence increases by several orders of magnitude the speed-up of the polling process by simply sending the command string as a whole directly to the PMAC using the raw hardware port. This was accomplished by marshaling the entire command string into a buffer which was then shipped down to the device driver via the ***MC_SendCommand()*** user-mode driver call shown in Figure 20. The marshaling process resulted in only one call to the kernel-mode driver to facilitate sending the entire command buffer to the PMAC instead of issuing multiple driver calls for each character of the command. This allows a higher performance polling paradigm then originally used because it lowers the level of polling integration to directly on top of the hardware instead of being context switched in the user-mode DLL. This results in one kernel driver call for each command invocation via a hardware-level BUS communication scheme not one call per character sent.

Figure 20 illustrates the ***MC_SendCommand()*** function that allows the application to send an entire command down to PMAC. The function is embedded into the user-mode driver and is accessible

[21] The original API, in this case *inp()* continued to be used as is; however, the functional part of the API was replaced with a call to a Windows NT/Win32 specific system call ***DeviceIoControl()***.

from application space using MCI or the low-level API. Notice the bolded IOCTL function code for sending a motion command to PMAC. This refers back to Figure 15 and the ***Dispatch()*** driver function where a request comes into the driver. This function illustrates many of the ideas necessary to provide user-mode application access to kernel-mode drivers. That is, notice in the function code that the Win32 API call to ***DeviceIoControl()*** is used. This method provides a standardized way of controlling the motion control device through the kernel-mode driver. In fact, this is the preferred method for providing portable device control and access across many of the newly emerging Microsoft operating systems.

4.3.3 Dual Port RAM Communications

The PMAC board provides an optional Dual Port RAM (random access memory that can be used for both transmit and receive operations) capability for extending the RAM addressed by the PMAC's on-board microprocessor, as well as other functions. The Dual-Port RAM effectively provides a very high-speed communications interface for both commanding the PMAC and setting and getting status information. This memory is typically used in two ways: first, it provides an additional addressing area to store and execute motion programs. Other functions provide for some of the RAM to be partitioned and used specifically for various status monitoring operations or high speed reading and writing of memory regions. The way in which dual-port memory is used can vary depending on the application requirements. In the case of the PMAC, application's used the memory regions provided by DLL's to access areas in clearly defined terms. That is, memory addresses in dual port memory are partitioned into application-specific areas that can only be used for certain functions such as monitoring the position of a motor axis or the speed of the motor. These high speed memory areas do not change.

Dual-Port RAM access to the application was provided in the device driver by using low-level Windows NT functions to map the address of the motion control board's Dual-Port RAM into the user's address space of the application/DLL. Figure 21 illustrates the NT function ***ZwMapViewOfSection()*** kernel call that maps the specified RAM address into the application/DLL space. Upon successful return from this call, the pointer ***VirtualDpramAddress*** is sent back to the calling application/DLL (through an I/O stack location specified in the IRP), providing the connection to the DLL for any future memory addressing and accessing. This allows an application/DLL to provide direct memory addressing capabilities by bypassing any NT-based filesystem security checks.

```
ntStatus = ZwMapViewOfSection (physicalMemoryHandle, (HANDLE) -1,
                &pDevInfo->VirtualDpramAddress, 0L, length,
                &viewBase, &length, ViewShare,
                0, PAGE_READWRITE | PAGE_NOCACHE);
if (!NT_SUCCESS(ntStatus))
{
    dprintf (("ZwMapViewOfSection failed"));
    ZwClose (physicalMemoryHandle);
}
```

Figure 21: Mapping the DPRAM Using *ZwMapViewOfSection()*.

Microsoft provides several basic example source modules illustrating this concept. With the user-space memory mapping modules from the MAPMEM DDK example, we were able to quickly map the PMAC onboard DPRAM into the user-mode DLL's that the applications could use.

Important reasons exist for why this direct approach would not be expected for use in the production environment. The memory mapping capability needs to be made more secure from application space. This is typically done by developing a user-mode driver that will map the RAM into NT's system space. Subsequent access to the memory mapped areas would be negotiated between the user-mode and kernel-mode drivers by sending messages from the application to the user-mode driver to initiate such delegated actions. However, the kernel-mode driver would in fact provide the actual accessing of memory on behalf of the user-mode application.

For completeness, we also included this secure capability within the device driver. The NT driver provides this capability by using a ***MmMapIoSpace()*** kernel call shown in Figure 22. Notice in Figure 22 that the pointer ***pDevInfo->FrameBase*** is used in a similar way that the I/O port address was mapped into the Window NT system space.

```
/* translate bus-relative frame buffer address using HAL, and map buffer into system
memory */

VOID MC_MapMemory(PDEVICE_INFO pDevInfo, PUCHAR Base, ULONG
Length)
{
   PHYSICAL_ADDRESS BaseAddress;
   PHYSICAL_ADDRESS MappedAddress;
   pDevInfo->FrameMemType = 0;    // memory space
   BaseAddress.LowPart = (ULONG) Base;
   BaseAddress.HighPart = 0;
   HalTranslateBusAddress(
         pDevInfo->BusType,
         pDevInfo->BusNumber,
         BaseAddress,
         &pDevInfo->FrameMemType,
         &MappedAddress);
   if (pDevInfo->FrameMemType == 0) {
      /* memory mapped i/o - map the physical address into system space*/
      pDevInfo->FrameBase = MmMapIoSpace(MappedAddress, Length, FALSE);
      dprintf2(("pmac dpram mapped at 0x%x", pDevInfo->FrameBase));
   } else {
      pDevInfo->FrameBase = (PUCHAR) MappedAddress.LowPart;
      dprintf(("pmac using raw address 0x%x", pDevInfo->FrameBase));
   }
   pDevInfo->FrameLength = Length;
}
```

Figure 22: Mapping DPRAM into NT System Space Using *MmMapIoSpace()*.

The complete NIST Windows NT device driver package was put together with the help of many software tools. These tools are part of the complete software development and build environment we used to produce the set of user and kernel-mode device drivers. These tools along with the development environment will be discussed to familiarize the reader with the process of creating, installing, configuring, and debugging Windows NT device drivers. The software development environment was used to develop, build, debug and test both the Windows NT user-mode multimedia device driver as well as the kernel-mode drivers.

4.4 Development Approach

To get familiar with the NT device driver environment, we used Microsoft's cookbook process for starting the development process. In short, the process merely guides the developer through a series of steps to take in order to build a skeletal driver environment using certain prerequisite functions. At this point the Microsoft Windows NT Software development Kit (SDK), the Device driver Development Kit (DDK), and a suitable compiler environment should all be installed properly to host the NT device driver build environment.

After compiling and linking the test driver, a small application is written that merely opens the device object associated with the driver and closes the driver object. This is a minimal driver, but after going through this process the developer becomes familiar with NT driver development including the build and debugging environments. After being satisfied that we had correctly installed the various kits to provide the build environment we wrote our own test driver as the Microsoft cookbook process suggested.

Feeling confident with the build and debug environment, Microsoft next recommends sifting through the various examples of device driver source code in the DDK to try to find a driver that may closely fit what you're trying to accomplish. Fortunately, we found a video capture hardware device driver that seemed to suit many of our needs; that is, it consisted of a multimedia-based user-mode driver and a paired kernel-mode NT device driver for the hardware capture board. Most of the time spent developing the driver was in retrofitting the existing driver to support the capabilities of the PMAC board.

The user-mode driver was also modified to support the MCI multimedia interface. We obtained a videodisk MCI source code example from the DDK to use as a template for this process. All PMAC device-level code for the driver was then written and integrated into the new driver shell. When a reasonable set of skeleton drivers for the PMAC was at hand, we used the build process described in the DDK documentation to compile, link, and assemble a driver installation package that was used to start initial debugging.

4.4.1 Driver Development Environment

Windows NT device driver development requires two Windows NT systems: one for the actual device driver development process (host) and one for debugging the executing driver code (the target). In addition, the second or target machine executes a kernel debugger application. The device driver development process introduces the notion of using a free (retail) and checked versions of the Windows NT operating system binaries.

A free build refers to the end-user version of the Windows NT operating system. This version of the Windows NT operating system is built using full optimization, all debugging asserts statements are disabled and debugging information is stripped out of the binary image. A free system or driver is smaller, faster, and uses less memory. The checked version of the Windows NT operating system allows testing and debugging for Windows NT system and driver developers. The checked build contains extra error checking, argument verification, and debugging information not found in the free build of Windows NT. A checked system or driver can help isolate and track down driver problems that cause unpredictable behavior, resulting in memory leaks or improper device configuration.

The checked build provides extra protection, and as a result it consumes more memory and disk space than the free build. System and driver performance is much slower because the executables contain symbolic debugging information. In addition, multiple code paths are executed due to parameter checking and debug output (diagnostic messages). Typically, driver development and debugging is done from a system running the free build. This host system is generally the more powerful of the two, with the retail (free) build of Windows NT installed along with the Win32 software development kit and the Windows NT device driver development kit. The SDK and DDK together provide the environment and build tools needed for device driver development in the Windows NT setting. The kits however, do not provide a compiler environment. Add-on 32-bit compilers from Borland or Microsoft can be used. However, to reduce integration issues with the Microsoft tools, usage of the Microsoft Visual C++ compiler version 2.00 (integrated C compiler version 9.00) or greater is highly recommended. Other third party compiler technology is not guaranteed to integrate well with the Microsoft hosted environment. The host system provides a stable environment for building the driver. The target system can be a less capable Windows NT machine running a minimal checked build installation.

Both the free and checked builds of Windows NT provide the hooks necessary for debug communication using a serial cable with a kernel debugger application running on a second Windows NT system. The host system runs the kernel debugger application and typically executes the free Windows NT kernel. The target system being debugged is running with kernel debug hooks enabled and may likewise be running either the free or checked build.

Driver development begins by testing and debugging a checked version of the driver using a system with a checked kernel. Later, a free version of the driver is tested and debugged using a system containing a free kernel. This step is important as the executing driver may behave differently in the two environments if timing or resource requirements in the driver become critical. Performance tuning of the driver should be done with the free build. Final testing and verification must be done on the retail (free) Windows NT system.

4.4.2 User-Mode Source-Level Debugging

Debugging the user-mode device driver and application software was done by using the inherent source-level debugging facilities provided in Microsoft Visual C++ Version 2.00. The MCITest application was initially executed to exercise the MCI portion of the driver. To initiate debugging, we would set breakpoints in the user-mode application to stop at a selected device driver API function. Doing this would allow us to step *into* the API call in the user-mode driver. Debugging the application and the user-mode device driver together using a source-level debugger provides tremendous capabilities. The application would call on the device driver in user-mode to carry out some sort of MCI or low-level API call and the driver could be stopped and the contents of the call captured. At this point the debugger in Visual C++ is used as the watch and display window host. Ultimately, the user-mode driver would make calls to the kernel-mode driver to complete the request for hardware access. When this would occur, a completely new set of interactions would happen requiring a new set of debugging tools. At this point, the ***Windbg*** SDK-based kernel debugger on the host machine would be used. This capability is discussed in the next section.

4.4.3 Kernel-Mode Source-Level Debugging

The user-mode driver must call upon the kernel-mode driver to accomplish any hardware level access. The hardware level access is only provided in kernel-mode and requires a different debugger. The Microsoft SDK provides a source-level debugger, ***Windbg*** (shown in Figure 23).

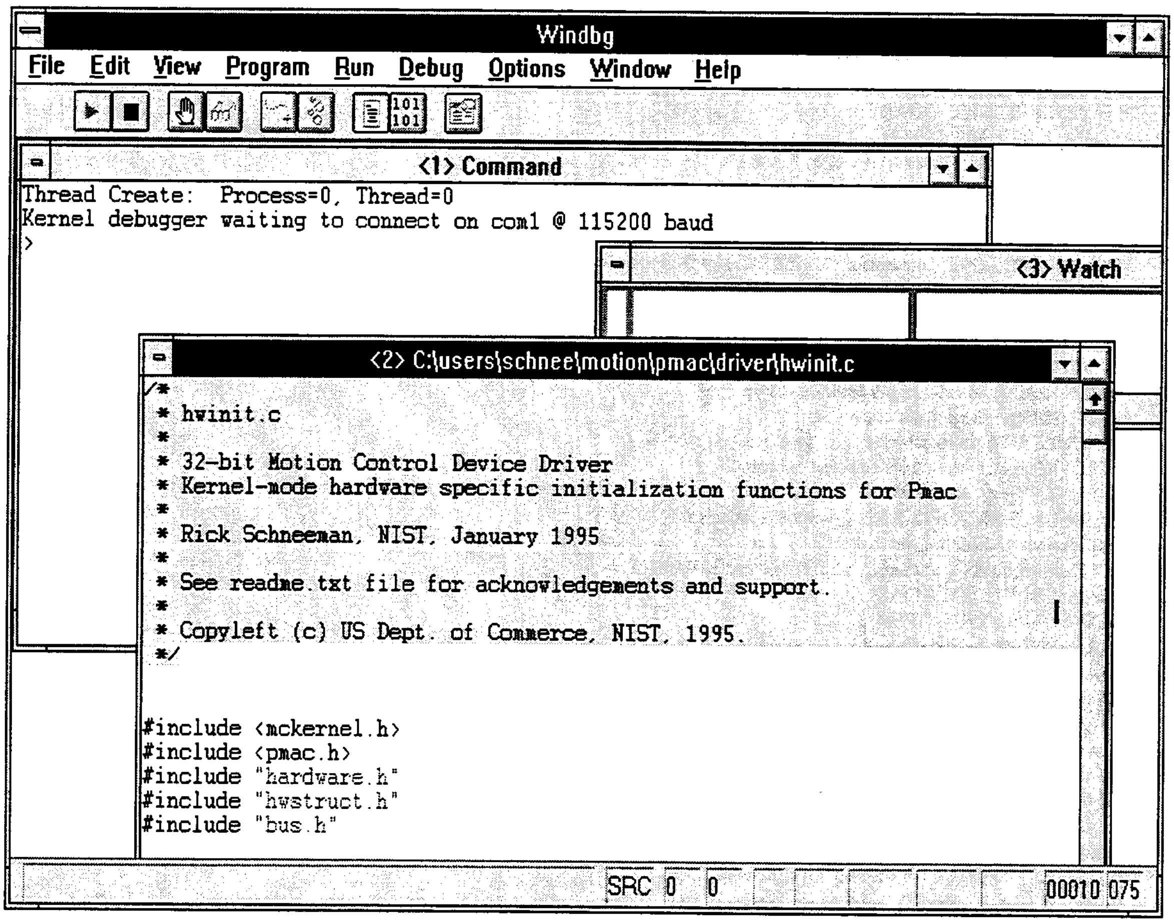

Figure 23: A Kernel-Mode Debug Session Using *WinDbg*.

Only starting recently (Windows NT Workstation Version 3.5) has this version of ***Windbg*** provided the capability to debug kernel-mode drivers at the source-level. This has been quite a boon to the Windows NT device driver development community. Machine level debugging of a kernel-mode driver is an extremely painful and arduous task. Now with the source-level capability, the user-mode debugging suite can be used in unison with the kernel-mode debugger ***WinDbg*** to provide a versatile platform for debugging. That is, on the target machine running the application, user-mode driver, and kernel-mode drivers the Microsoft Visual C++ debugging environment executes. On the host machine connected via a high-speed serial port to the target, the host executes the ***Windbg*** program. Working together the application can call the user-mode driver which is viewed on the Visual C++ display, while the user-mode driver calls the kernel driver with its display status shown in the ***Windbg*** screen on the host. This is a powerful capability that enhances the Windows NT device driver development process. The ***WinDbg*** command window shown in Figure 23 typically provides the developer with insight into what sort of exchanges are taking place between the debugger and debuggee (through the serial connection). In addition, full source-level debugging of the kernel-mode driver is provided. During debugging, a log file is optionally kept for tracing the command window at a later time. An example of a partial log file is shown in Figure 24.

```
Thread Create:  Process=0, Thread=0
Kernel debugger waiting to connect on com1 @ 115200 baud
Kernel Debugger connection established on com1 @ 115200 baud
Kernel Version 807 Checked loaded @ 0x80100000
Module Load: NTOSKRNL.EXE  (symbol loading deferred)
Module Load: C:\NT35\symbols\EXE\NTOSKRNL.DBG  (symbols converted & loaded)
Module Load: C:\NT35\symbols\DLL\PMAC.dll  (symbols loaded)
PMAC (USER):DRV_LOAD
PMAC (USER):DRV_OPEN
PMAC (USER):Configuration open in progress
PMAC (USER):DRV_INSTALL
PMAC (USER):DRV_QUERYCONFIGURE
PMAC (USER):DRV_CONFIGURE
Module Load: C:\NT35\symbols\SYS\PMAC.sys  (symbols loaded)
PMAC (KERN): pmac at port 0x210, int 10, dpram 0xd4000
PMAC (KERN): 16 ports at 0x210 (i/o space)
PMAC (KERN): mapping dpram into system
PMAC (KERN): pmac dpram mapped at 0xfc895000
PMAC (KERN): all interrupts enabled: 0x0
PMAC (KERN): board init complete
PMAC (KERN): connecting to interrupt 0xa
PMAC (KERN): timeout waiting for interrupt!
PMAC (KERN): driver loaded
Module Load: MCITEST.EXE  (symbols loaded)
MCITest: started (debug level 1)
MCITest: MCITEST starting... module handle is 400000H
PMAC (USER):DRV_LOAD
PMAC (USER):DRV_OPEN
PMAC (USER):device opened
PMAC (USER):MCI_OPEN_DRIVER...
PMAC (USER):MCI_INTR_INIT
PMAC (USER):current priority class: 32
PMAC (USER):new priority class: 256
PMAC (USER):worker 122 starting
PMAC (KERN): IOCTL_MOTION_INTR_INIT
PMAC (USER):interrupt init ok
PMAC (USER):interrupt request 0..7 queued
PMAC (USER):thread priority set for time critical
PMAC (USER):waiting for completion event
PMAC (USER):completion signalled
PMAC (USER):MCI_SERVO_JOG
PMAC (KERN):  161 usec/interrupt
MCITest: roundtrip message latency: 200 (ms)
PMAC (USER):interrupt request 0 completed
PMAC (USER):interrupt request 0 queue
```

Figure 24: Trace of *Windbg* Output During Kernel-Mode Debugging.

The important exchanges include debugging print statements that the developer embeds into the driver software. By tracing these print statements, a much more efficient means of arriving at a predetermined state in the driver can be reached. In the log file of Figure 24, notice that the messages are coming from both the user-mode and kernel-mode driver. This is evident by the use of the USER and KERN parenthetical expressions in the log file indicating the source of the message. In addition, notice that the symbols for both the user-mode driver and the kernel-mode driver needed to be loaded early on in the kernel-mode debug initialization process (i.e., PMAC.dll, and PMAC.sys). This is required in order to do source-level debugging of the two driver images during debugging.

4.5 MCITest — An Application for Multimedia Driver Interface Testing

In order to exercise the user-mode multimedia driver we developed, an application that used both the message and string interfaces of the new motion-based MCI command set was required. An application that minimally tests the core MCI command set (located on the Microsoft SDK CDROM), was selected as the template to exercise our extended MCI command set. The resulting application MCITest, shown in Figure 25, was modified to include new user interface characteristics that would provide the capability to: **(1)** exercise the NIST-extended MCI command set, **(2)** utilize both user and kernel-mode driver interfaces, **(3)** test the new application-level interrupt scheme, and to **(4)** provide an entry point into the user-mode dynamic link library (an ultimately the kernel-mode driver), which would allow source-level debugging.

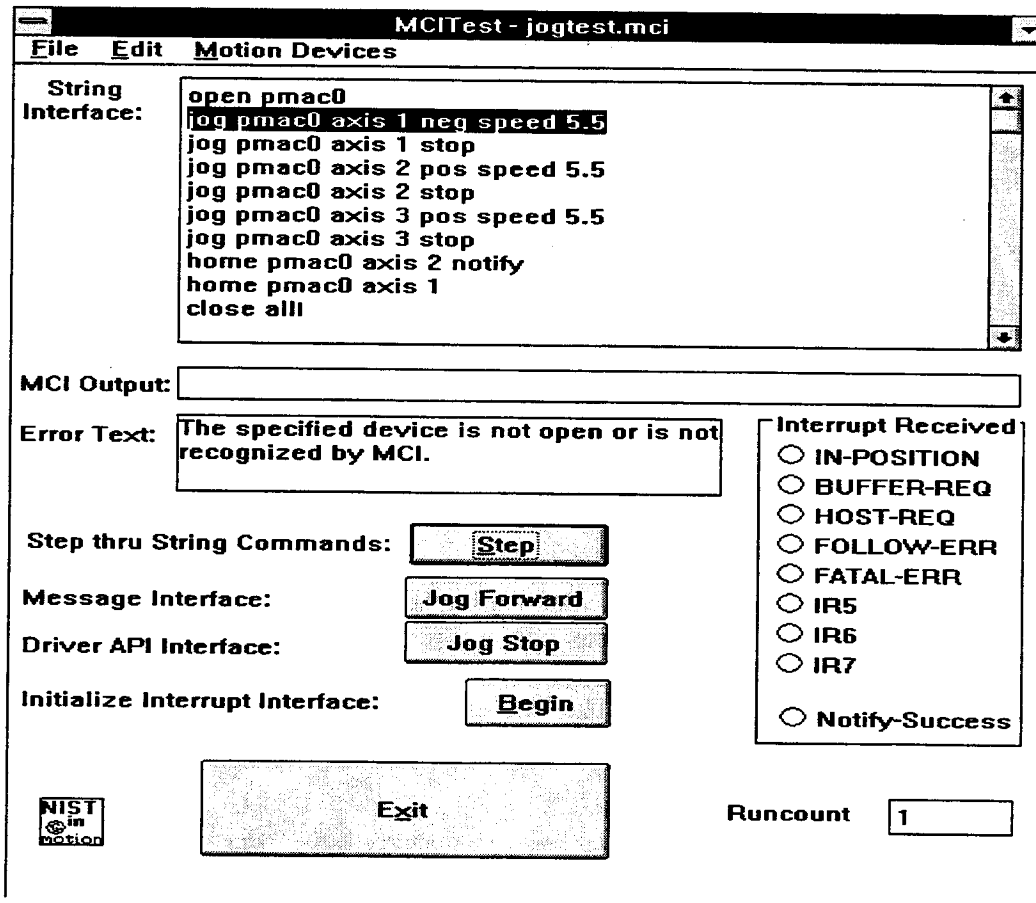

Figure 25: MCITest: Exercises the MCI-based Motion Control Interface.

Because the legacy interrupt scheme required application-based interrupt notification, we needed a way to initialize the interrupt interface through MCI. This was implemented through a push button on the interface that when selected issued a MCI command to the driver to begin interrupt initialization (Section 4.3.1.2). A special notification area in the user interface of MCITest shown in Figure 25 illustrates the reception and notification of any interrupt received from the PMAC.

We installed and loaded the Windows NT device driver pair on a computer in our laboratory that controls a machine tool it is connected to. The machine tool is a large Hardinge SuperSlant Lathe, capable of very high-speed, high precision machining processes. This machine was retrofitted with a PMAC controller from another study.

The MCITest program was designed to test the device driver interface on this machine tool. Using MCITest, the user would simply start a motor axis on the machine tool jogging (rotating). After the axis begins turning, the user would either click on the stop jog button with the mouse, or enter the stop jog command as a string to MCI text input window shown in Figure 25. When the user activates the button representing the jog stop command or enters the stop string in the text input window, a connection between the MCITest application and the user-mode device driver is made. The user-mode driver in turn interacts with the kernel-mode device driver to complete the request. Therefore the MCITest user interface actually connects to code that interacts with the user-mode driver.

After the user clicks on a specific button, the code in the application would execute until a breakpoint at that function would be set; thereby allowing the user-mode debugger (Visual C++ debugger on the target machine) to begin the source-level debugging process[22]. Eventually, the user-mode driver would call the kernel-mode driver, thus initiating the source-level debug process on the host NT machine using ***WinDbg***. This entire process takes place between two different machines in real-time via a high-speed serial port connection.

The text input window in Figure 25 also illustrates some of the custom as well as standard MCI-based string commands that can be issued to the user-mode multimedia device driver including *jog*, *stop*, *home*, *open*, and *close*. The MCI string commands to *jog* or *home* the specific axis all include additional parameters as strings to MCI. Specific commands require different associated parameters to quantify the command to MCI. For instance, the *jog* command needs to know what device to jog, what axis or motor to rotate, at what speed, and in what direction of rotation— negative or positive direction. Recall that the custom commands and parameters were constructed from the custom command table that we developed for the MCI-based control of motion devices.

MCI specific flags or parameters in the string are also illustrated in Figure 25. That is, the notify parameter is used to provide application-based notification when the command string has completed processing. The all parameter on the *close* command is also a MCI specific parameter that simply provides a convenient way of closing all the devices in the motion control-based MCI environment that have been opened. The MCITest application provided a convenient interface for us to test and debug the set of Windows NT device drivers.

---

22 Debugging tools allow developers to insert breakpoints into the debug source code. The breakpoints represent locations in the source code that the user wishes the program execution to momentarily stop. When program execution halts at these points, programmers can use other debugger capabilities to watch specific variables, step through instructions one at a time, or to change certain values in memory locations. The main reason we set breakpoints in the MCITest application was to provide a passageway into the user-mode driver.

5. Summary

We present in this document our current techniques and approaches to implementing device drivers for the control of motion devices in the Windows NT environment. We chose multimedia as the venue and specifically Microsoft's MCI interface because the device-independent control of peripheral devices was the key guiding principle that we wanted to strive for during this process.

The MCI interface generated from this research will benefit industry by allowing users of multiple motion control hardware greater access to a variety of hardware solutions while providing software developers the same application programming interface to motion control. Developers can provide their own custom applications for any product that supports Microsoft's multimedia environment. In addition, all hardware access required by the user-mode multimedia driver is facilitated by the use of an associated Windows NT kernel-mode device-dependent driver in the set. Together they provide complete PMAC-based motion control services using Microsoft standard multimedia accessing methods.

Integrators may extend these device drivers for custom applications just as we extended the existing MCI interface for unsupported motion-based functionality. We specifically used the MCI for motion related messaging to fulfill the requirements for our study; however, the messaging framework does not preclude supplanting other motion control application domains with additional messaging contexts. Indeed, the applicability of this type of framework is that it can be extended to support message requirements for other motion control related applications readily.

Because all hardware dependence has been isolated at the kernel level, other motion control board manufacturers need only reengineer it to provide hardware services based on their particular communication strategies. This effort should be minimal, due to motion control board manufacturers utilize similar communication approaches such as interrupt-driven, BUS-based, or Dual-Ported RAM techniques.

The set of device drivers developed at NIST were tested out using two different environments. During the early development effort, the device drivers were debugged using a hardware demonstration box developed by Delta Tau that safely provided actual motion-based simulation using small motors and supporting electronics. When the device driver development effort stabilized, these drivers were successfully installed, tested, and used in a working machine shop setting using live machine tools.

As of this writing, Delta Tau Data Systems has begun development on a beta version of Windows NT device drivers for future products. Many of the concepts developed as part of the NIST device driver effort have been incorporated into this release.

6. References

[1] Microsoft Windows NT Device Driver Kit, *Kernel-mode Driver Design Guide*, Microsoft Press, 1993.

[2] Microsoft Win32 Programmer's Reference Guide. *Volume 2: System Services, Multimedia, Extensions, and Application Notes*. Microsoft Press, 1993.

[3] Microsoft Windows NT Device Driver Kit. *Win32 Subsystem Driver Design Guide*. Microsoft Press, 1993.

[4] PMAC Users Manual, Firmware Version 1.13, Delta Tau Data Systems, Inc., December 1992.

Acknowledgment

The examples from the Microsoft SDK & DDK were invaluable in developing the Motion Control device driver set. In addition, I would especially like to thank Dimitri S. Dimitri, Dennis Smith, and Allen Segall of Delta Tau Data Systems for providing valuable source code, feedback, reference materials, and technical support during this process. Finally, I would like to thank the reviewers of this document for their advice and encouragement during this manuscript development process.

Obtaining the Software

This work was done under the auspices of the United Stated Department of Commerce, National Institute of Standards and Technology — a U.S. Government Agency. Therefore, the software developed at NIST cannot be copyrighted in the U.S. We have placed the source code for the device driver in the Public Domain and is governed by the GNU Copyleft regulations and those set forth by the Secretary of Commerce, U.S. Department of Commerce. Version 1.0 of the software is available through anonymous ftp at daedalus.aptd.nist.gov (Internet Address: 129.6.36.21, for non-naming sites) in the pub/motion directory as motion.exe (a self-extracting ZIP file). Although not discussed in this paper, there is also a limited device driver for the Linux operating system (Slackware version 2.2). A graphical user interface-based Tcl/Tk application is provided with the Linux driver to exercise the PMAC BUS interface. The Linux driver is located in the same directory as the NT driver under the filename motion.tgz (GNU zipped and tarred).

About the Author

Richard Schneeman is a Computer Scientist in the Manufacturing Engineering Laboratory (MEL) at NIST. He is a member of the Automated Production Technology Division where currently he is involved with integrating sensor-based technology with next generation control networking protocols and standards. Other research and development projects he has been involved with at NIST include: multimedia computing, microkernel operating systems, advanced distributed systems, high-speed communication networks, and protocol conformance testing.